O'REILLY®
オライリー・ジャパン

TURING 图灵程序设计丛书

深度学习入门

基于Python的理论与实现

Deep Learning from Scratch



[日] 斋藤康毅 著
陆宇杰 译

中国工信出版集团 人民邮电出版社
POSTS & TELECOM PRESS

作者介绍

斋藤康毅

日本东京工业大学毕业，并完成东京大学研究生院课程。现从事计算机视觉与机器学习相关的研究和开发工作。是*Introducing Python*、*Python in Practice*、*The Elements of Computing Systems*、*Building Machine Learning Systems with Python*的日文版译者。

译者介绍

陆宇杰

众安科技NLP算法工程师。主要研究方向为自然语言处理及其应用，对图像识别、机器学习、深度学习等领域有密切关注。Python爱好者。

数字版权声明

TURING 图灵程序设计丛书

深度学习入门
基于Python的理论与实现

Deep Learning from Scratch

[日] 斋藤康毅 著
陆宇杰 译

O'REILLY®

Beijing · Boston · Farnham · Sebastopol · Tokyo
O'Reilly Japan, Inc. 授权人民邮电出版社出版

人民邮电出版社
北　京

图书在版编目（CIP）数据

深度学习入门 ：基于Python的理论与实现 / （日）
斋藤康毅著 ；陆宇杰译. -- 北京 ：人民邮电出版社，
2018.7
（图灵程序设计丛书）
ISBN 978-7-115-48558-8

Ⅰ. ①深… Ⅱ. ①斋… ②陆… Ⅲ. ①软件工具－程
序设计 Ⅳ. ①TP311.561

中国版本图书馆CIP数据核字(2018)第112509号

内 容 提 要

本书是深度学习真正意义上的入门书，深入浅出地剖析了深度学习的原理和相关技术。书中使用Python 3，尽量不依赖外部库或工具，带领读者从零创建一个经典的深度学习网络，使读者在此过程中逐步理解深度学习。书中不仅介绍了深度学习和神经网络的概念、特征等基础知识，对误差反向传播法、卷积神经网络等也有深入讲解，此外还介绍了学习相关的实用技巧，自动驾驶、图像生成、强化学习等方面的应用，以及为什么加深层可以提高识别精度等“为什么”的问题。

本书适合深度学习初学者阅读，也可作为高校教材使用。

◆ 著　　　　[日] 斋藤康毅
　译　　　　陆宇杰
　责任编辑　杜晓静
　执行编辑　刘香娣
　责任印制　周昇亮

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
　邮编　100164　　电子邮件　315@ptpress.com.cn
　网址　http://www.ptpress.com.cn
　北京　　　　　　　　印刷

◆ 开本：880×1230　1/32
　印张：9.625
　字数：300千字　　　　　2018年7月第1版
　印数：1 - 4 000册　　　2018年7月北京第1次印刷

著作权合同登记号　图字：01-2017-0526号

定价：59.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广登字20170147号

版 权 声 明

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 *Make* 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

译者序

深度学习的浪潮已经汹涌澎湃了一段时间了，市面上相关的图书也已经出版了很多。其中，既有知名学者伊恩·古德费洛（Ian Goodfellow）等人撰写的系统介绍深度学习基本理论的《深度学习》，也有各种介绍深度学习框架的使用方法的入门书。你可能会问，现在再出一本关于深度学习的书，是不是“为时已晚”？其实并非如此，因为本书考察深度学习的角度非常独特，它的出版可以说是“千呼万唤始出来”。

本书最大的特点是“剖解”了深度学习的底层技术。正如美国物理学家理查德·费曼（Richard Phillips Feynman）所说：“What I cannot create, I do not understand.”只有创造一个东西，才算真正弄懂了一个问题。本书就是教你如何创建深度学习模型的一本书。并且，本书不使用任何现有的深度学习框架，尽可能仅使用最基本的数学知识和Python库，从零讲解深度学习核心问题的数学原理，从零创建一个经典的深度学习网络。

本书的日文版曾一度占据了东京大学校内书店（本乡校区）理工类图书的畅销书榜首。各类读者阅读本书，均可有所受益。对于非AI方向的技术人员，本书将大大降低入门深度学习的门槛；对于在校的大学生、研究生，本书不失为学习深度学习的一本好教材；即便是对于在工作中已经熟练使用框架开发各类深度学习模型的读者，也可以从本书中获得新的体会。

本书从开始翻译到出版，前前后后历时一年之久。译者翻译时力求忠于原文，表达简练。为了保证翻译质量，每翻译完一章后，译者都会放置一段

时间，再重新检查一遍。图灵公司的专业编辑们又进一步对译稿进行了全面细致的校对，提出了许多宝贵意见，在此表示感谢。但是，由于译者才疏学浅，书中难免存在一些错误或疏漏，恳请读者批评指正，以便我们在重印时改正。

最后，希望本书的出版能为国内的AI技术社区添砖加瓦！

陆宇杰

2018年2月 上海

前言

科幻电影般的世界已经变成了现实——人工智能战胜过日本将棋、国际象棋的冠军，最近甚至又打败了围棋冠军；智能手机不仅可以理解人们说的话，还能在视频通话中进行实时的“机器翻译”；配备了摄像头的“自动防撞的车”保护着人们的生命安全，自动驾驶技术的实用化也为期不远。环顾我们的四周，原来被认为只有人类才能做到的事情，现在人工智能都能毫无差错地完成，甚至试图超越人类。因为人工智能的发展，我们所处的世界正在逐渐变成一个崭新的世界。

在这个发展速度惊人的世界背后，深度学习技术在发挥着重要作用。对于深度学习，世界各地的研究人员不吝褒奖之辞，称赞其为革新性技术，甚至有人认为它是几十年才有一次的突破。实际上，深度学习这个词经常出现在报纸和杂志中，备受关注，就连一般大众也都有所耳闻。

本书就是一本以深度学习为主题的书，目的是让读者尽可能深入地理解深度学习的技术。因此，本书提出了“从零开始”这个概念。

本书的特点是通过实现深度学习的过程，来逼近深度学习的本质。通过实现深度学习的程序，尽可能无遗漏地介绍深度学习相关的技术。另外，本书还提供了实际可运行的程序，供读者自己进行各种各样的实验。

为了实现深度学习，我们需要经历很多考验，花费很长时间，但是相应地也能学到和发现很多东西。而且，实现深度学习的过程是一个有趣的、令

人兴奋的过程。希望读者通过这一过程可以熟悉深度学习中使用的技术，并能从中感受到快乐。

目前，深度学习活跃在世界上各个地方。在几乎人手一部的智能手机中、开启自动驾驶的汽车中、为Web服务提供动力的服务器中，深度学习都在发挥着作用。此时此刻，就在很多人没有注意到的地方，深度学习正在默默地发挥着其功能。今后，深度学习势必将更加活跃。为了让读者理解深度学习的相关技术，感受到深度学习的魅力，笔者写下了本书。

本书的理念

本书是一本讲解深度学习的书，将从最基础的内容开始讲起，逐一介绍理解深度学习所需的知识。书中尽可能用平实的语言来介绍深度学习的概念、特征、工作原理等内容。不过，本书并不是只介绍技术的概要，而是旨在让读者更深入地理解深度学习。这是本书的特色之一。

那么，怎么才能更深入地理解深度学习呢？在笔者看来，最好的办法就是亲自实现。从零开始编写可实际运行的程序，一边看源代码，一边思考。笔者坚信，这种做法对正确理解深度学习（以及那些看上去很高级的技术）是很重要的。这里用了“从零开始”一词，表示我们将尽可能地不依赖外部的现成品（库、工具等）。也就是说，本书的目标是，尽量不使用内容不明的黑盒，而是从自己能理解的最基础的知识出发，一步一步地实现最先进的深度学习技术。并通过这一实现过程，使读者加深对深度学习的理解。

如果把本书比作一本关于汽车的书，那么本书并不会教你怎么开车，其着眼点不是汽车的驾驶方法，而是要让读者理解汽车的原理。为了让读者理解汽车的结构，必须打开汽车的引擎盖，把零件一个一个地拿在手里观察，并尝试操作它们。之后，用尽可能简单的形式提取汽车的本质，并组装汽车模型。本书的目标是，通过制造汽车模型的过程，让读者感受到自己可以实际制造出汽车，并在这一过程中熟悉汽车相关的技术。

为了实现深度学习，本书使用了Python这一编程语言。Python非常受欢迎，初学者也能轻松使用。Python尤其适合用来制作样品（原型），使用

Python可以立刻尝试突然想到的东西，一边观察结果，一边进行各种各样的实验。本书将在讲解深度学习理论的同时，使用Python实现程序，进行各种实验。

> 在光看数学式和理论说明无法理解的情况下，可以尝试阅读源代码并运行，很多时候思路都会变得清晰起来。对数学式感到困惑时，就阅读源代码来理解技术的流程，这样的事情相信很多人都经历过。本书通过实际实现（落实到代码）来理解深度学习，是一本强调"工程"的书。书中会出现很多数学式，但同时也会有很多程序员视角的源代码。

本书面向的读者

本书旨在让读者通过实际动手操作来深入理解深度学习。为了明确本书的读者对象，这里将本书涉及的内容列举如下。

- 使用Python，尽可能少地使用外部库，从零开始实现深度学习的程序。
- 为了让Python的初学者也能理解，介绍Python的使用方法。
- 提供实际可运行的Python源代码，同时提供可以让读者亲自实验的学习环境。
- 从简单的机器学习问题开始，最终实现一个能高精度地识别图像的系统。
- 以简明易懂的方式讲解深度学习和神经网络的理论。
- 对于误差反向传播法、卷积运算等乍一看很复杂的技术，使读者能够在实现层面上理解。
- 介绍一些学习深度学习时有用的实践技巧，如确定学习率的方法、权重的初始值等。
- 介绍最近流行的Batch Normalization、Dropout、Adam等，并进行实现。
- 讨论为什么深度学习表现优异、为什么加深层能提高识别精度、为什么隐藏层很重要等问题。
- 介绍自动驾驶、图像生成、强化学习等深度学习的应用案例。

本书不面向的读者

明确本书不适合什么样的读者也很重要。为此，这里将本书不会涉及的内容列举如下。

- 不介绍深度学习相关的最新研究进展。
- 不介绍Caffe、TensorFlow、Chainer等深度学习框架的使用方法。
- 不介绍深度学习的详细理论，特别是神经网络相关的详细理论。
- 不详细介绍用于提高识别精度的参数调优相关的内容。
- 不会为了实现深度学习的高速化而进行GPU相关的实现。
- 本书以图像识别为主题，不涉及自然语言处理或者语音识别的例子。

综上，本书不涉及最新研究和理论细节。但是，读完本书之后，读者应该有能力进一步去阅读最新的论文或者神经网络相关的理论方面的技术书。

本书以图像识别为主题，主要学习使用深度学习进行图像识别时所需的技术。自然语言处理或者语音识别等不是本书的讨论对象。

本书的阅读方法

学习新知识时，只听别人讲解的话，有时会无法理解，或者会立刻忘记。正如“不闻不若闻之，闻之不若见之，见之不若知之，知之不若行之”[1]，在学习新东西时，没有什么比实践更重要了。本书在介绍某个主题时，都细心地准备了一个可以实践的场所——能够作为程序运行的源代码。

本书会提供Python源代码，读者可以自己动手实际运行这些源代码。在阅读源代码的同时，可以尝试去实现一些自己想到的东西，以确保真正

① 出自荀子《儒效篇》。

理解了。另外，读者也可以使用本书的源代码，尝试进行各种实验，反复试错。

本书将沿着“理论说明”和“Python实现”两个路线前进。因此，建议读者准备好编程环境。本书可以使用Windows、Mac、Linux中的任何一个系统。关于Python的安装和使用方法将在第1章介绍。另外，本书中用到的程序可以从以下网址下载。

http://www.ituring.com.cn/book/1921

让我们开始吧

通过前面的介绍，希望读者了解本书大概要讲的内容，产生继续阅读的兴趣。

最近出现了很多深度学习相关的库，任何人都可以方便地使用。实际上，使用这些库的话，可以轻松地运行深度学习的程序。那么，为什么我们还要特意花时间从零开始实现深度学习呢？一个理由就是，在制作东西的过程中可以学到很多。

在制作东西的过程中，会进行各种各样的实验，有时也会卡住，抱着脑袋想为什么会这样。这种费时的工作对深刻理解技术而言是宝贵的财富。像这样认真花费时间获得的知识在使用现有的库、阅读最新的文章、创建原创的系统时都大有用处。而且最重要的是，制作本身就是一件快乐的事情。（还需要快乐以外的其他什么理由吗？）

既然一切都准备好了，下面就让我们踏上实现深度学习的旅途吧！

表述规则

本书在表述上采用如下规则。

粗体字（Bold）

用来表示新引入的术语、强调的要点以及关键短语。

等宽字（`Constant Width`）

用来表示下面这些信息：程序代码、命令、序列、组成元素、语句选项、分支、变量、属性、键值、函数、类型、类、命名空间、方法、模块、属性、参数、值、对象、事件、事件处理器、XML标签、HTML标签、宏、文件的内容、来自命令行的输出等。若在其他地方引用了以上这些内容（如变量、函数、关键字等），也会使用该格式标记。

等宽粗体字（`Constant Width Bold`）

用来表示用户输入的命令或文本信息。在强调代码的作用时也会使用该格式标记。

等宽斜体字（*`Constant Width Italic`*）

用来表示必须根据用户环境替换的字符串。



用来表示提示、启发以及某些值得深究的内容的补充信息。



表示程序库中存在的bug或时常会发生的问题等警告信息，引起读者对该处内容的注意。

读者意见与咨询

虽然笔者已经尽最大努力对本书的内容进行了验证与确认，但仍不免在某些地方出现错误或者容易引起误解的表达等，给读者的理解带来困扰。如果读者遇到这些问题，请及时告知，我们在本书重印时会将其改正，在此先表示不胜感激。与此同时，也希望读者能够为本书将来的修订提出中肯的建议。本书编辑部的联系方式如下。

株式会社O'Reilly Japan

电子邮件 japan@oreilly.co.jp

本书的主页地址如下。

http://www.ituring.com.cn/book/1583

http://www.oreilly.co.jp/books/9784873117584（日语）

https://github.com/oreilly-japan/deep-learning-from-scratch

关于O'Reilly的其他信息，可以访问下面的O'Reilly主页查看。

http://www.oreilly.com/（英语）

http://www.oreilly.co.jp/（日语）

致谢

首先，笔者要感谢推动了深度学习相关技术（机器学习、计算机科学等）发展的研究人员和工程师。本书的完成离不开他们的研究工作。其次，笔者还要感谢在图书或网站上公开有用信息的各位同仁。其中，斯坦福大学的CS231n[5]公开课慷慨提供了很多有用的技术和信息，笔者从中学到了很多东西。

在本书执笔过程中，曾受到下列人士的帮助：teamLab公司的加藤哲朗、喜多慎弥、飞永由夏、中野皓太、中村将达、林辉大、山本辽；Top Studio公司的武藤健志、增子萌；Flickfit公司的野村宪司；得克萨斯大学奥斯汀分校JSPS海外特别研究员丹野秀崇。他们阅读了本书原稿，提出了很多宝贵的建议，在此深表谢意。另外，需要说明的是，本书中存在的不足或错误均是笔者的责任。

最后，还要感谢O'Reilly Japan的宫川直树，在从本书的构想到完成的大约一年半的时间里，宫川先生一直支持着笔者。非常感谢！

2016年9月1日

斋藤康毅

第1章 Python入门

Python这一编程语言已经问世20多年了，在这期间，Python不仅完成了自身的进化，还获得了大量的用户。现在，Python作为最具人气的编程语言，受到了许多人的喜爱。

接下来我们将使用Python实现深度学习系统。不过在这之前，本章将简单地介绍一下Python，看一下它的使用方法。已经掌握了Python、NumPy、Matplotlib等知识的读者，可以跳过本章，直接阅读后面的章节。

1.1 Python是什么

Python是一个简单、易读、易记的编程语言，而且是开源的，可以免费地自由使用。Python可以用类似英语的语法编写程序，编译起来也不费力，因此我们可以很轻松地使用Python。特别是对首次接触编程的人士来说，Python是最合适不过的语言。事实上，很多高校和大专院校的计算机课程均采用Python作为入门语言。

此外，使用Python不仅可以写出可读性高的代码，还可以写出性能高（处理速度快）的代码。在需要处理大规模数据或者要求快速响应的情况下，使用Python可以稳妥地完成。因此，Python不仅受到初学者的喜爱，同时也受到专业人士的喜爱。实际上，Google、Microsoft、Facebook等战斗在IT行业最前沿的企业也经常使用Python。

再者，在科学领域，特别是在机器学习、数据科学领域，Python也被大量使用。Python除了高性能之外，凭借着NumPy、SciPy等优秀的数值计算、统计分析库，在数据科学领域占有不可动摇的地位。深度学习的框架中也有很多使用Python的场景，比如Caffe、TensorFlow、Chainer、Theano等著名的深度学习框架都提供了Python接口。因此，学习Python对使用深度学习框架大有益处。

综上，Python是最适合数据科学领域的编程语言。而且，Python具有受众广的优秀品质，从初学者到专业人士都在使用。因此，为了完成本书的从零开始实现深度学习的目标，Python可以说是最合适的工具。

1.2 Python的安装

下面，我们首先将Python安装到当前环境（电脑）上。这里说明一下安装时需要注意的一些地方。

1.2.1 Python版本

Python有Python 2.x和Python 3.x两个版本。如果我们调查一下目前Python的使用情况，会发现除了最新的版本3.x以外，旧的版本2.x仍在被大量使用。因此，在安装Python时，需要慎重选择安装Python的哪个版本。这是因为两个版本之间没有兼容性（严格地讲，是没有“向后兼容性”），也就是说，会发生用Python 3.x写的代码不能被Python 2.x执行的情况。本书中使用Python 3.x，只安装了Python 2.x的读者建议另外安装一下Python 3.x。

1.2.2 使用的外部库

本书的目标是从零开始实现深度学习。因此，除了NumPy库和Matplotlib库之外，我们极力避免使用外部库。之所以使用这两个库，是因为它们可以有效地促进深度学习的实现。

NumPy是用于数值计算的库，提供了很多高级的数学算法和便利的数组（矩阵）操作方法。本书中将使用这些便利的方法来有效地促进深度学习的实现。

Matplotlib是用来画图的库。使用Matplotlib能将实验结果可视化，并在视觉上确认深度学习运行期间的数据。

本书将使用下列编程语言和库。

- Python 3.*x*（2016年8月时的最新版本是3.5）
- NumPy
- Matplotlib

下面将为需要安装Python的读者介绍一下Python的安装方法。已经安装了Python的读者，请跳过这一部分内容。

1.2.3 Anaconda发行版

Python的安装方法有很多种，本书推荐使用Anaconda这个发行版。发行版集成了必要的库，使用户可以一次性完成安装。Anaconda是一个侧重于数据分析的发行版，前面说的NumPy、Matplotlib等有助于数据分析的库都包含在其中[①]。

如前所述，本书将使用Python 3.x版本，因此Anaconda发行版也要安装3.x的版本。请读者从官方网站下载与自己的操作系统相应的发行版，然后安装。

① Anaconda作为一个针对数据分析的发行版，包含了许多有用的库，而本书中实际上只会使用其中的NumPy库和Matplotlib库。因此，如果想保持轻量级的开发环境，单独安装这两个库也是可以的。——译者注

1.3 Python解释器

完成Python的安装后，要先确认一下Python的版本。打开终端（Windows中的命令行窗口），输入`python --version`命令，该命令会输出已经安装的Python的版本信息。

```
$ python --version
Python 3.4.1 :: Anaconda 2.1.0 (x86_64)
```

如上所示，显示了Python 3.4.1（根据实际安装的版本，版本号可能不同），说明已正确安装了Python 3.x。接着输入`python`，启动Python解释器。

```
$ python
Python 3.4.1 |Anaconda 2.1.0 (x86_64)| (default, Sep 10 2014, 17:24:09)
[GCC 4.2.1 (Apple Inc. build 5577)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Python解释器也被称为"对话模式"，用户能够以和Python对话的方式进行编程。比如，当用户询问"1 + 2等于几？"的时候，Python解释器会回答"3"，所谓对话模式，就是指这样的交互。现在，我们实际输入一下看看。

```
>>> 1 + 2
3
```

Python解释器可以像这样进行对话式（交互式）的编程。下面，我们使用这个对话模式，来看几个简单的Python编程的例子。

1.3.1 算术计算

加法或乘法等算术计算，可按如下方式进行。

```
>>> 1 - 2
-1
>>> 4 * 5
20
```

```
>>> 7 / 5
1.4
>>> 3 ** 2
9
```

`*`表示乘法，`/`表示除法，`**`表示乘方（`3**2`是3的2次方）。另外，在Python 2.x中，整数除以整数的结果是整数，比如，$7 \div 5$的结果是1。但在Python 3.x中，整数除以整数的结果是小数（浮点数）。

1.3.2 数据类型

编程中有**数据类型**（data type）这一概念。数据类型表示数据的性质，有整数、小数、字符串等类型。Python中的`type()`函数可以用来查看数据类型。

```
>>> type(10)
<class 'int'>
>>> type(2.718)
<class 'float'>
>>> type("hello")
<class 'str'>
```

根据上面的结果可知，`10`是`int`类型（整型），`2.718`是`float`类型（浮点型），`"hello"`是`str`（字符串）类型。另外，“类型”和“类”这两个词有时用作相同的意思。这里，对于输出结果`<class 'int'>`，可以将其解释成“`10`是`int`类（类型）”。

1.3.3 变量

可以使用`x`或`y`等字母定义**变量**（variable）。此外，可以使用变量进行计算，也可以对变量赋值。

```
>>> x = 10   # 初始化
>>> print(x) # 输出x
10
>>> x = 100  # 赋值
>>> print(x)
100
```

```
>>> y = 3.14
>>> x * y
314.0
>>> type(x * y)
<class 'float'>
```

Python是属于“动态类型语言”的编程语言，所谓动态，是指变量的类型是根据情况自动决定的。在上面的例子中，用户并没有明确指出“x的类型是int（整型）”，是Python根据x被初始化为10，从而判断出x的类型为int的。此外，我们也可以看到，整数和小数相乘的结果是小数（数据类型的自动转换）。另外，“#”是注释的意思，它后面的文字会被Python忽略。

1.3.4 列表

除了单一的数值，还可以用列表（数组）汇总数据。

```
>>> a = [1, 2, 3, 4, 5] # 生成列表
>>> print(a)  # 输出列表的内容
[1, 2, 3, 4, 5]
>>> len(a)    # 获取列表的长度
5
>>> a[0]      # 访问第一个元素的值
1
>>> a[4]
5
>>> a[4] = 99 # 赋值
>>> print(a)
[1, 2, 3, 4, 99]
```

元素的访问是通过a[0]这样的方式进行的。[]中的数字称为索引（下标），索引从0开始（索引0对应第一个元素）。此外，Python的列表提供了**切片**（slicing）这一便捷的标记法。使用切片不仅可以访问某个值，还可以访问列表的子列表（部分列表）。

```
>>> print(a)
[1, 2, 3, 4, 99]
>>> a[0:2] # 获取索引为0到2（不包括2！）的元素
[1, 2]
>>> a[1:]  # 获取从索引为1的元素到最后一个元素
[2, 3, 4, 99]
```

```
>>> a[:3]  # 获取从第一个元素到索引为3(不包括3！)的元素
[1, 2, 3]
>>> a[:-1] # 获取从第一个元素到最后一个元素的前一个元素之间的元素
[1, 2, 3, 4]
>>> a[:-2] # 获取从第一个元素到最后一个元素的前二个元素之间的元素
[1, 2, 3]
```

进行列表的切片时，需要写成a[0:2]这样的形式。a[0:2]用于取出从索引为0的元素到索引为2的元素的前一个元素之间的元素。另外，索引-1对应最后一个元素，-2对应最后一个元素的前一个元素。

1.3.5 字典

列表根据索引，按照$0, 1, 2, \cdots$的顺序存储值，而字典则以键值对的形式存储数据。字典就像《新华字典》那样，将单词和它的含义对应着存储起来。

```
>>> me = {'height':180} # 生成字典
>>> me['height']        # 访问元素
180
>>> me['weight'] = 70   # 添加新元素
>>> print(me)
{'height': 180, 'weight': 70}
```

1.3.6 布尔型

Python中有bool型。bool型取True或False中的一个值。针对bool型的运算符包括and、or和not(针对数值的运算符有+、-、*、/等，根据不同的数据类型使用不同的运算符)。

```
>>> hungry = True     # 饿了?
>>> sleepy = False    # 困了?
>>> type(hungry)
<class 'bool'>
>>> not hungry
False
>>> hungry and sleepy # 饿并且困
False
>>> hungry or sleepy  # 饿或者困
True
```

1.3.7 if语句

根据不同的条件选择不同的处理分支时可以使用`if/else`语句。

```
>>> hungry = True
>>> if hungry:
...     print("I'm hungry")
...
I'm hungry
>>> hungry = False
>>> if hungry:
...     print("I'm hungry") # 使用空白字符进行缩进
... else:
...     print("I'm not hungry")
...     print("I'm sleepy")
...
I'm not hungry
I'm sleepy
```

Python中的空白字符具有重要的意义。上面的`if`语句中，`if hungry:`下面的语句开头有4个空白字符。它是缩进的意思，表示当前面的条件(`if hungry`)成立时，此处的代码会被执行。这个缩进也可以用tab表示，Python中推荐使用空白字符。

Python使用空白字符表示缩进。一般而言，每缩进一次，使用4个空白字符。

1.3.8 for 语句

进行循环处理时可以使用`for`语句。

```
>>> for i in [1, 2, 3]:
...     print(i)
...
1
2
3
```

这是输出列表`[1,2,3]`中的元素的例子。使用`for … in … :`语句结构，

可以按顺序访问列表等数据集合中的各个元素。

1.3.9 函数

可以将一连串的处理定义成**函数**(function)。

```
>>> def hello():
...     print("Hello World!")
...
>>> hello()
Hello World!
```

此外，函数可以取参数。

```
>>> def hello(object):
...     print("Hello " + object + "!")
...
>>> hello("cat")
Hello cat!
```

另外，字符串的拼接可以使用`+`。

关闭Python解释器时，Linux或Mac OS X的情况下输入Ctrl-D(按住Ctrl，再按D键)；Windows的情况下输入Ctrl-Z，然后按Enter键。

1.4 Python脚本文件

到目前为止，我们看到的都是基于Python解释器的例子。Python解释器能够以对话模式执行程序，非常便于进行简单的实验。但是，想进行一连串的处理时，因为每次都需要输入程序，所以不太方便。这时，可以将Python程序保存为文件，然后(集中地)运行这个文件。下面，我们来看一个Python脚本文件的例子。

1.4.1 保存为文件

打开文本编辑器，新建一个`hungry.py`的文件。`hungry.py`只包含下面一行语句。

```
print("I'm hungry!")
```

接着，打开终端（Windows 中的命令行窗口），移至 hungry.py 所在的位置。然后，将 hungry.py 文件名作为参数，运行 python 命令。这里假设 hungry.py 在 ~/deep-learning-from-scratch/ch01 目录下（在本书提供的源代码中，hungry.py 文件位于 ch01 目录下）。

```
$ cd ~/deep-learning-from-scratch/ch01 # 移动目录
$ python hungry.py
I'm hungry!
```

这样，使用 python hungry.py 命令就可以执行这个 Python 程序了。

1.4.2 类

前面我们了解了 int 和 str 等数据类型（通过 type() 函数可以查看对象的类型）。这些数据类型是"内置"的数据类型，是 Python 中一开始就有的数据类型。现在，我们来定义新的类。如果用户自己定义类的话，就可以自己创建数据类型。此外，也可以定义原创的方法（类的函数）和属性。

Python 中使用 class 关键字来定义类，类要遵循下述格式（模板）。

```
class 类名:
    def __init__(self, 参数, …): # 构造函数
       ...
    def 方法名1(self, 参数, …):   # 方法1
       ...
    def 方法名2(self, 参数, …):   # 方法2
       ...
```

这里有一个特殊的 __init__ 方法，这是进行初始化的方法，也称为**构造函数**（constructor），只在生成类的实例时被调用一次。此外，在方法的第一个参数中明确地写入表示自身（自身的实例）的 self 是 Python 的一个特点（学过其他编程语言的人可能会觉得这种写 self 的方式有一点奇怪）。

下面我们通过一个简单的例子来创建一个类。这里将下面的程序保存为 man.py。

```
class Man:
    def __init__(self, name):
        self.name = name
        print("Initialized!")

    def hello(self):
        print("Hello " + self.name + "!")

    def goodbye(self):
        print("Good-bye " + self.name + "!")

m = Man("David")
m.hello()
m.goodbye()
```

从终端运行`man.py`。

```
$ python man.py
Initialized!
Hello David!
Good-bye David!
```

这里我们定义了一个新类`Man`。上面的例子中，类`Man`生成了实例（对象）`m`。

类`Man`的构造函数（初始化方法）会接收参数`name`，然后用这个参数初始化实例变量`self.name`。**实例变量**是存储在各个实例中的变量。Python中可以像`self.name`这样，通过在`self`后面添加属性名来生成或访问实例变量。

1.5 NumPy

在深度学习的实现中，经常出现数组和矩阵的计算。NumPy的数组类（`numpy.array`）中提供了很多便捷的方法，在实现深度学习时，我们将使用这些方法。本节我们来简单介绍一下后面会用到的NumPy。

1.5.1 导入NumPy

NumPy是外部库。这里所说的“外部”是指不包含在标准版Python中。因此，我们首先要导入NumPy库。

```
>>> import numpy as np
```

Python中使用import语句来导入库。这里的import numpy as np，直译的话就是“将numpy作为np导入”的意思。通过写成这样的形式，之后NumPy相关的方法均可通过np来调用。

1.5.2 生成NumPy数组

要生成NumPy数组，需要使用np.array()方法。np.array()接收Python列表作为参数，生成NumPy数组(numpy.ndarray)。

```
>>> x = np.array([1.0, 2.0, 3.0])
>>> print(x)
[ 1. 2. 3.]
>>> type(x)
<class 'numpy.ndarray'>
```

1.5.3 NumPy 的算术运算

下面是NumPy数组的算术运算的例子。

```
>>> x = np.array([1.0, 2.0, 3.0])
>>> y = np.array([2.0, 4.0, 6.0])
>>> x + y  # 对应元素的加法
array([ 3.,  6., 9.])
>>> x - y
array([ -1.,  -2., -3.])
>>> x * y  # element-wise product
array([  2.,   8.,  18.])
>>> x / y
array([ 0.5,  0.5,  0.5])
```

这里需要注意的是，数组x和数组y的元素个数是相同的(两者均是元素个数为3的一维数组)。当x和y的元素个数相同时，可以对各个元素进行算术运算。如果元素个数不同，程序就会报错，所以元素个数保持一致非常重要。另外，“对应元素的”的英文是element-wise，比如“对应元素的乘法”就是element-wise product。

NumPy数组不仅可以进行element-wise运算，也可以和单一的数值(标量)

组合起来进行运算。此时，需要在NumPy数组的各个元素和标量之间进行运算。这个功能也被称为广播（详见后文）。

```
>>> x = np.array([1.0, 2.0, 3.0])
>>> x / 2.0
array([ 0.5,  1. ,  1.5])
```

1.5.4 NumPy的N维数组

NumPy不仅可以生成一维数组（排成一列的数组），也可以生成多维数组。比如，可以生成如下的二维数组（矩阵）。

```
>>> A = np.array([[1, 2], [3, 4]])
>>> print(A)
[[1 2]
 [3 4]]
>>> A.shape
(2, 2)
>>> A.dtype
dtype('int64')
```

这里生成了一个2×2的矩阵A。另外，矩阵A的形状可以通过shape查看，矩阵元素的数据类型可以通过dtype查看。下面，我们来看一下矩阵的算术运算。

```
>>> B = np.array([[3, 0],[0, 6]])
>>> A + B
array([[ 4,  2],
       [ 3, 10]])
>>> A * B
array([[ 3,  0],
       [ 0, 24]])
```

和数组的算术运算一样，矩阵的算术运算也可以在相同形状的矩阵间以对应元素的方式进行。并且，也可以通过标量（单一数值）对矩阵进行算术运算。这也是基于广播的功能。

```
>>> print(A)
[[1 2]
 [3 4]]
```

```
>>> A * 10
array([[ 10, 20],
       [ 30, 40]])
```



NumPy数组（np.array）可以生成N维数组，即可以生成一维数组、二维数组、三维数组等任意维数的数组。数学上将一维数组称为**向量**，将二维数组称为**矩阵**。另外，可以将一般化之后的向量或矩阵等统称为**张量**（tensor）。本书基本上将二维数组称为“矩阵”，将三维数组及三维以上的数组称为“张量”或“多维数组”。

1.5.5 广播

NumPy中，形状不同的数组之间也可以进行运算。之前的例子中，在2×2的矩阵A和标量10之间进行了乘法运算。在这个过程中，如图1-1所示，标量10被扩展成了2×2的形状，然后再与矩阵A进行乘法运算。这个巧妙的功能称为**广播**（broadcast）。

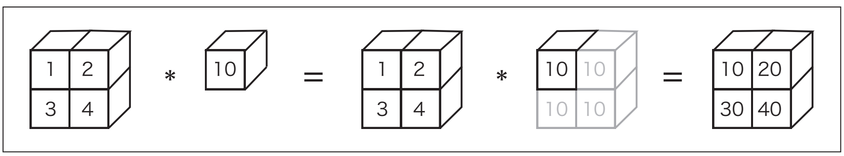


图1-1 广播的例子：标量10被当作2×2的矩阵

我们通过下面这个运算再来看一个广播的例子。

```
>>> A = np.array([[1, 2], [3, 4]])
>>> B = np.array([10, 20])
>>> A * B
array([[ 10, 40],
       [ 30, 80]])
```

在这个运算中，如图1-2所示，一维数组B被“巧妙地”变成了和二位数组A相同的形状，然后再以对应元素的方式进行运算。

综上，因为NumPy有广播功能，所以不同形状的数组之间也可以顺利地进行运算。

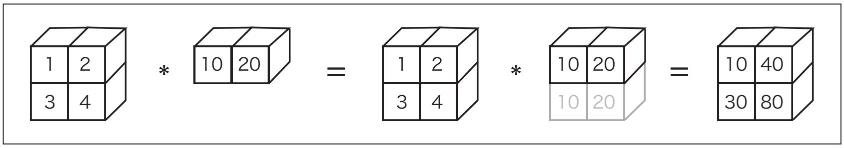


图1-2　广播的例子2

1.5.6　访问元素

元素的索引从0开始。对各个元素的访问可按如下方式进行。

```
>>> X = np.array([[51, 55], [14, 19], [0, 4]])
>>> print(X)
[[51 55]
 [14 19]
 [ 0  4]]
>>> X[0]    # 第0行
array([51, 55])
>>> X[0][1] # (0,1)的元素
55
```

也可以使用for语句访问各个元素。

```
>>> for row in X:
...     print(row)
...
[51 55]
[14 19]
[0 4]
```

除了前面介绍的索引操作，NumPy还可以使用数组访问各个元素。

```
>>> X = X.flatten()         # 将X转换为一维数组
>>> print(X)
[51 55 14 19  0  4]
>>> X[np.array([0, 2, 4])] # 获取索引为0、2、4的元素
array([51, 14,  0])
```

运用这个标记法，可以获取满足一定条件的元素。例如，要从X中抽出大于15的元素，可以写成如下形式。

```
>>> X > 15
array([ True,  True, False,  True, False, False], dtype=bool)
>>> X[X>15]
array([51, 55, 19])
```

对NumPy数组使用不等号运算符等（上例中是`X > 15`），结果会得到一个布尔型的数组。上例中就是使用这个布尔型数组取出了数组的各个元素（取出`True`对应的元素）。

> Python等动态类型语言一般比C和C++等静态类型语言（编译型语言）运算速度慢。实际上，如果是运算量大的处理对象，用C/C++写程序更好。为此，当Python中追求性能时，人们会用C/C++来实现处理的内容。Python则承担"中间人"的角色，负责调用那些用C/C++写的程序。NumPy中，主要的处理也都是通过C或C++实现的。因此，我们可以在不损失性能的情况下，使用Python便利的语法。

1.6 Matplotlib

在深度学习的实验中，图形的绘制和数据的可视化非常重要。Matplotlib是用于绘制图形的库，使用Matplotlib可以轻松地绘制图形和实现数据的可视化。这里，我们来介绍一下图形的绘制方法和图像的显示方法。

1.6.1 绘制简单图形

可以使用`matplotlib`的`pyplot`模块绘制图形。话不多说，我们来看一个绘制sin函数曲线的例子。

```
import numpy as np
import matplotlib.pyplot as plt

# 生成数据
x = np.arange(0, 6, 0.1) # 以0.1为单位，生成0到6的数据
y = np.sin(x)

# 绘制图形
```

```
plt.plot(x, y)
plt.show()
```

这里使用NumPy的arange方法生成了[0, 0.1, 0.2, ···, 5.8, 5.9]的数据，将其设为x。对x的各个元素，应用NumPy的sin函数np.sin()，将x、y的数据传给plt.plot方法，然后绘制图形。最后，通过plt.show()显示图形。运行上述代码后，就会显示图1-3所示的图形。

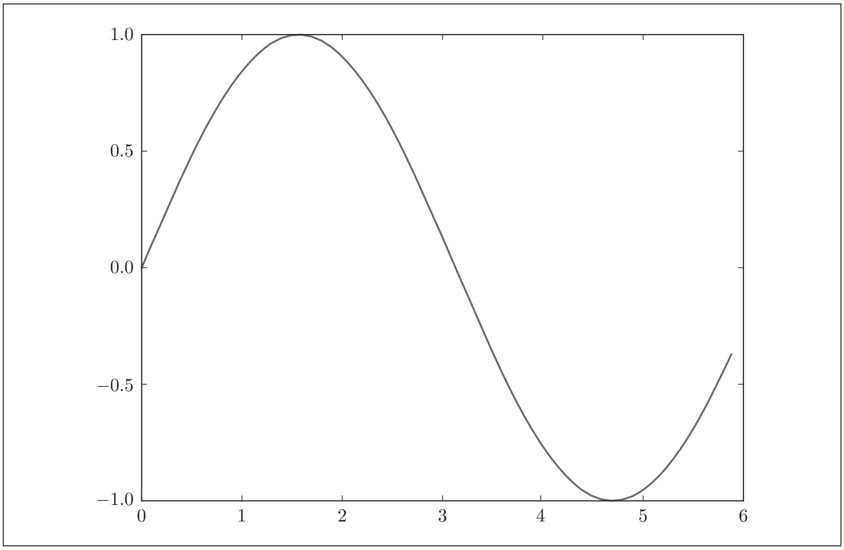


图1-3 sin函数的图形

1.6.2 pyplot的功能

在刚才的sin函数的图形中，我们尝试追加cos函数的图形，并尝试使用pyplot的添加标题和x轴标签名等其他功能。

```
import numpy as np
import matplotlib.pyplot as plt

# 生成数据
x = np.arange(0, 6, 0.1) # 以0.1为单位，生成0到6的数据
y1 = np.sin(x)
```

```
y2 = np.cos(x)

# 绘制图形
plt.plot(x, y1, label="sin")
plt.plot(x, y2, linestyle = "--", label="cos") # 用虚线绘制
plt.xlabel("x") # x轴标签
plt.ylabel("y") # y轴标签
plt.title('sin & cos') # 标题
plt.legend()
plt.show()
```

结果如图1-4所示，我们看到图的标题、轴的标签名都被标出来了。

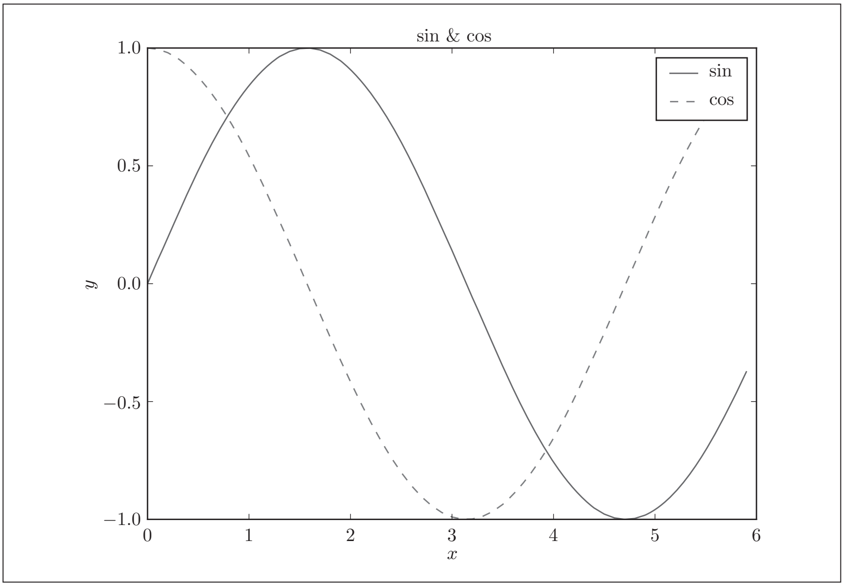


图1-4 sin函数和cos函数的图形

1.6.3 显示图像

pyplot中还提供了用于显示图像的方法imshow()。另外，可以使用matplotlib.image模块的imread()方法读入图像。下面我们来看一个例子。

```
import matplotlib.pyplot as plt
from matplotlib.image import imread
```

```
img = imread('lena.png') # 读入图像(设定合适的路径!)
plt.imshow(img)

plt.show()
```

运行上述代码后，会显示图1-5所示的图像。

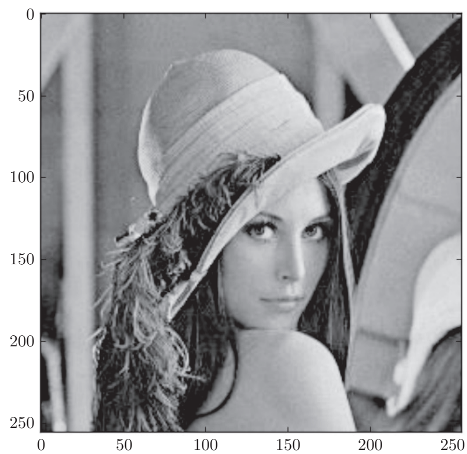


图1-5 显示图像

这里，我们假定图像lena.png在当前目录下。读者根据自己的环境，可能需要变更文件名或文件路径。另外，本书提供的源代码中，在dataset目录下有样本图像lena.png。比如，在通过Python解释器从ch01目录运行上述代码的情况下，将图像的路径'lena.png'改为'../dataset/lena.png'，即可正确运行。

1.7 小结

本章重点介绍了实现深度学习(神经网络)所需的编程知识，以为学习深度学习做好准备。从下一章开始，我们将通过使用Python实际运行代码，逐步了解深度学习。

本章只介绍了关于Python的最低限度的知识，想进一步了解Python的读者，可以参考下面这些图书。首先推荐《Python语言及其应用》[1]一书。这是一本详细介绍从Python编程的基础到应用的实践性的入门书。关于NumPy，《利用Python进行数据分析》[2]一书中进行了简单易懂的总结。此外，“Scipy Lecture Notes”[3]这个网站上也有以科学计算为主题的NumPy和Matplotlib的详细介绍，有兴趣的读者可以参考。

下面，我们来总结一下本章所学的内容，如下所示。

本章所学的内容

- Python是一种简单易记的编程语言。
- Python是开源的，可以自由使用。
- 本书中将使用Python 3.x实现深度学习。
- 本书中将使用NumPy和Matplotlib这两种外部库。
- Python有“解释器”和“脚本文件”两种运行模式。
- Python能够将一系列处理集成为函数或类等模块。
- NumPy中有很多用于操作多维数组的便捷方法。

第2章 感知机

本章将介绍**感知机**[①]（perceptron）这一算法。感知机是由美国学者Frank Rosenblatt在1957年提出来的。为何我们现在还要学习这一很久以前就有的算法呢？因为感知机也是作为神经网络（深度学习）的起源的算法。因此，学习感知机的构造也就是学习通向神经网络和深度学习的一种重要思想。

本章我们将简单介绍一下感知机，并用感知机解决一些简单的问题。希望读者通过这个过程能熟悉感知机。

2.1 感知机是什么

感知机接收多个输入信号，输出一个信号。这里所说的“信号”可以想象成电流或河流那样具备“流动性”的东西。像电流流过导线，向前方输送电子一样，感知机的信号也会形成流，向前方输送信息。但是，和实际的电流不同的是，感知机的信号只有“流/不流”（1/0）两种取值。在本书中，0对应“不传递信号”，1对应“传递信号”。

图2-1是一个接收两个输入信号的感知机的例子。x_1、x_2是输入信号，y是输出信号，w_1、w_2是权重（w是weight的首字母）。图中的○称为“神经元”或者“节点”。输入信号被送往神经元时，会被分别乘以固定的权重

① 严格地讲，本章中所说的感知机应该称为“人工神经元”或“朴素感知机”，但是因为很多基本的处理都是共通的，所以这里就简单地称为“感知机”。

(w_1x_1、w_2x_2)。神经元会计算传送过来的信号的总和，只有当这个总和超过了某个界限值时，才会输出1。这也称为“神经元被激活”。这里将这个界限值称为**阈值**，用符号θ表示。

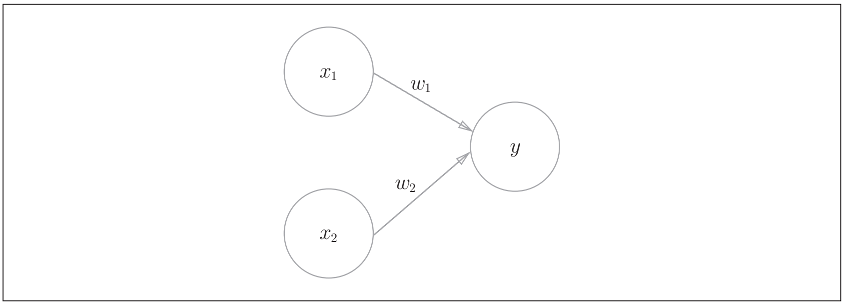


图2-1 有两个输入的感知机

感知机的运行原理只有这些！把上述内容用数学式来表示，就是式(2.1)。

$$y = \begin{cases} 0 & (w_1x_1 + w_2x_2 \leqslant \theta) \\ 1 & (w_1x_1 + w_2x_2 > \theta) \end{cases} \tag{2.1}$$

感知机的多个输入信号都有各自固有的权重，这些权重发挥着控制各个信号的重要性的作用。也就是说，权重越大，对应该权重的信号的重要性就越高。



权重相当于电流里所说的电阻。电阻是决定电流流动难度的参数，电阻越低，通过的电流就越大。而感知机的权重则是值越大，通过的信号就越大。不管是电阻还是权重，在控制信号流动难度(或者流动容易度)这一点上的作用都是一样的。

2.2 简单逻辑电路

2.2.1 与门

现在让我们考虑用感知机来解决简单的问题。这里首先以逻辑电路为题材来思考一下与门（AND gate）。与门是有两个输入和一个输出的门电路。图2-2这种输入信号和输出信号的对应表称为“真值表”。如图2-2所示，与门仅在两个输入均为1时输出1，其他时候则输出0。

x_1	x_2	y
0	0	0
1	0	0
0	1	0
1	1	1

图2-2　与门的真值表

下面考虑用感知机来表示这个与门。需要做的就是确定能满足图2-2的真值表的w_1、w_2、θ的值。那么，设定什么样的值才能制作出满足图2-2的条件的感知机呢？

实际上，满足图2-2的条件的参数的选择方法有无数多个。比如，当$(w_1, w_2, \theta) = (0.5, 0.5, 0.7)$时，可以满足图2-2的条件。此外，当$(w_1, w_2, \theta)$为$(0.5, 0.5, 0.8)$或者$(1.0, 1.0, 1.0)$时，同样也满足与门的条件。设定这样的参数后，仅当$x_1$和$x_2$同时为1时，信号的加权总和才会超过给定的阈值$\theta$。

2.2.2 与非门和或门

接着，我们再来考虑一下与非门（NAND gate）。NAND是Not AND的

意思，与非门就是颠倒了与门的输出。用真值表表示的话，如图2-3所示，仅当x_1和x_2同时为1时输出0，其他时候则输出1。那么与非门的参数又可以是什么样的组合呢？

x_1	x_2	y
0	0	1
1	0	1
0	1	1
1	1	0

图2-3 与非门的真值表

要表示与非门，可以用$(w_1, w_2, \theta) = (-0.5, -0.5, -0.7)$这样的组合（其他的组合也是无限存在的）。实际上，只要把实现与门的参数值的符号取反，就可以实现与非门。

接下来看一下图2-4所示的或门。或门是“只要有一个输入信号是1，输出就为1”的逻辑电路。那么我们来思考一下，应该为这个或门设定什么样的参数呢？

x_1	x_2	y
0	0	0
1	0	1
0	1	1
1	1	1

图2-4 或门的真值表

这里决定感知机参数的并不是计算机，而是我们人。我们看着真值表这种“训练数据”，人工考虑(想到)了参数的值。而机器学习的课题就是将这个决定参数值的工作交由计算机自动进行。**学习**是确定合适的参数的过程，而人要做的是思考感知机的构造(模型)，并把训练数据交给计算机。

如上所示，我们已经知道使用感知机可以表示与门、与非门、或门的逻辑电路。这里重要的一点是：与门、与非门、或门的感知机构造是一样的。实际上，3个门电路只有参数的值(权重和阈值)不同。也就是说，相同构造的感知机，只需通过适当地调整参数的值，就可以像“变色龙演员”表演不同的角色一样，变身为与门、与非门、或门。

2.3　感知机的实现

2.3.1　简单的实现

现在，我们用Python来实现刚才的逻辑电路。这里，先定义一个接收参数`x1`和`x2`的`AND`函数。

```
def AND(x1, x2):
    w1, w2, theta = 0.5, 0.5, 0.7
    tmp = x1*w1 + x2*w2
    if tmp <= theta:
        return 0
    elif tmp > theta:
        return 1
```

在函数内初始化参数`w1`、`w2`、`theta`，当输入的加权总和超过阈值时返回`1`，否则返回`0`。我们来确认一下输出结果是否如图2-2所示。

```
AND(0, 0) # 输出0
AND(1, 0) # 输出0
AND(0, 1) # 输出0
AND(1, 1) # 输出1
```

果然和我们预想的输出一样！这样我们就实现了与门。按照同样的步骤，

也可以实现与非门和或门，不过让我们来对它们的实现稍作修改。

2.3.2 导入权重和偏置

刚才的与门的实现比较直接、容易理解，但是考虑到以后的事情，我们将其修改为另外一种实现形式。在此之前，首先把式(2.1)的θ换成$-b$，于是就可以用式(2.2)来表示感知机的行为。

$$y = \begin{cases} 0 & (b + w_1x_1 + w_2x_2 \leqslant 0) \\ 1 & (b + w_1x_1 + w_2x_2 > 0) \end{cases} \tag{2.2}$$

式(2.1)和式(2.2)虽然有一个符号不同，但表达的内容是完全相同的。此处，b称为**偏置**，w_1和w_2称为**权重**。如式(2.2)所示，感知机会计算输入信号和权重的乘积，然后加上偏置，如果这个值大于0则输出1，否则输出0。下面，我们使用NumPy，按式(2.2)的方式实现感知机。在这个过程中，我们用Python的解释器逐一确认结果。

```
>>> import numpy as np
>>> x = np.array([0, 1])     # 输入
>>> w = np.array([0.5, 0.5]) # 权重
>>> b = -0.7                 # 偏置
>>> w*x
array([ 0. ,  0.5])
>>> np.sum(w*x)
0.5
>>> np.sum(w*x) + b
-0.19999999999999996   # 大约为-0.2(由浮点小数造成的运算误差)
```

如上例所示，在NumPy数组的乘法运算中，当两个数组的元素个数相同时，各个元素分别相乘，因此`w*x`的结果就是它们的各个元素分别相乘(`[0, 1] * [0.5, 0.5] => [0, 0.5]`)。之后，`np.sum(w*x)`再计算相乘后的各个元素的总和。最后再把偏置加到这个加权总和上，就完成了式(2.2)的计算。

2.3.3 使用权重和偏置的实现

使用权重和偏置，可以像下面这样实现与门。

```
def AND(x1, x2):
    x = np.array([x1, x2])
    w = np.array([0.5, 0.5])
    b = -0.7
    tmp = np.sum(w*x) + b
    if tmp <= 0:
       return 0
    else:
       return 1
```

这里把$-\theta$命名为偏置b，但是请注意，偏置和权重w_1、w_2的作用是不一样的。具体地说，w_1和w_2是控制输入信号的重要性的参数，而偏置是调整神经元被激活的容易程度（输出信号为1的程度）的参数。比如，若b为-0.1，则只要输入信号的加权总和超过0.1，神经元就会被激活。但是如果b为-20.0，则输入信号的加权总和必须超过20.0，神经元才会被激活。像这样，偏置的值决定了神经元被激活的容易程度。另外，这里我们将w_1和w_2称为权重，将b称为偏置，但是根据上下文，有时也会将b、w_1、w_2这些参数统称为权重。



偏置这个术语，有“穿木屐”[①]的效果，即在没有任何输入时（输入为0时），给输出穿上多高的木屐（加上多大的值）的意思。实际上，在式(2.2)的$b + w_1x_1 + w_2x_2$的计算中，当输入x_1和x_2为0时，只输出偏置的值。

接着，我们继续实现与非门和或门。

```
def NAND(x1, x2):
    x = np.array([x1, x2])
    w = np.array([-0.5, -0.5]) # 仅权重和偏置与AND不同！
    b = 0.7
    tmp = np.sum(w*x) + b
    if tmp <= 0:
        return 0
    else:
        return 1

def OR(x1, x2):
```

① 因为木屐的底比较厚，穿上它后，整个人也会显得更高。——译者注

```
    x = np.array([x1, x2])
    w = np.array([0.5, 0.5]) # 仅权重和偏置与AND不同!
    b = -0.2
    tmp = np.sum(w*x) + b
    if tmp <= 0:
        return 0
    else:
        return 1
```

我们在2.2节介绍过，与门、与非门、或门是具有相同构造的感知机，区别只在于权重参数的值。因此，在与非门和或门的实现中，仅设置权重和偏置的值这一点和与门的实现不同。

2.4 感知机的局限性

到这里我们已经知道，使用感知机可以实现与门、与非门、或门三种逻辑电路。现在我们来考虑一下异或门（XOR gate）。

2.4.1 异或门

异或门也被称为**逻辑异或**电路。如图2-5所示，仅当x_1或x_2中的一方为1时，才会输出1（“异或”是拒绝其他的意思）。那么，要用感知机实现这个异或门的话，应该设定什么样的权重参数呢？

x_1	x_2	y
0	0	0
1	0	1
0	1	1
1	1	0

图2-5 异或门的真值表

实际上，用前面介绍的感知机是无法实现这个异或门的。为什么用感知机可以实现与门、或门，却无法实现异或门呢？下面我们尝试通过画图来思考其中的原因。

首先，我们试着将或门的动作形象化。或门的情况下，当权重参数 $(b, w_1, w_2) = (-0.5, 1.0, 1.0)$ 时，可满足图 2-4 的真值表条件。此时，感知机可用下面的式（2.3）表示。

$$y = \begin{cases} 0 & (-0.5 + x_1 + x_2 \leqslant 0) \\ 1 & (-0.5 + x_1 + x_2 > 0) \end{cases} \tag{2.3}$$

式（2.3）表示的感知机会生成由直线 $-0.5 + x_1 + x_2 = 0$ 分割开的两个空间。其中一个空间输出 1，另一个空间输出 0，如图 2-6 所示。

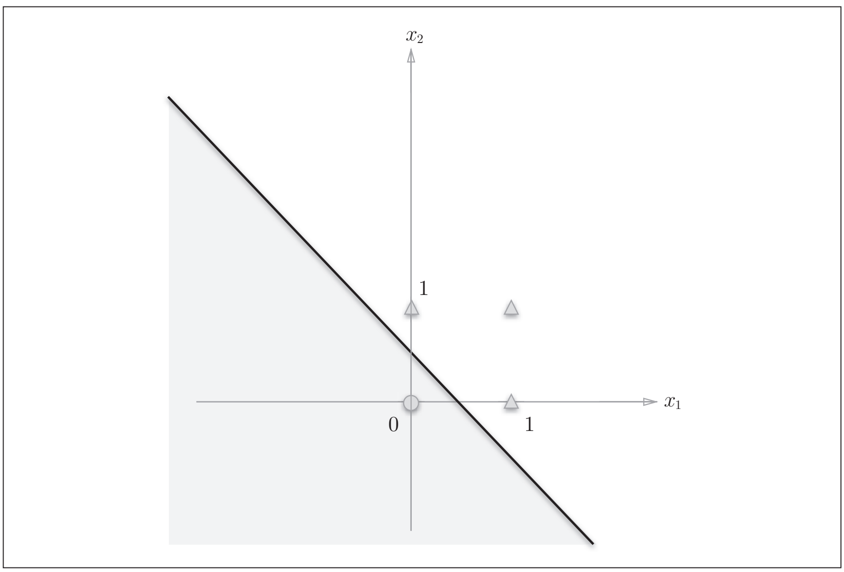


图 2-6　感知机的可视化：灰色区域是感知机输出 0 的区域，这个区域与或门的性质一致

或门在 $(x_1, x_2) = (0, 0)$ 时输出 0，在 (x_1, x_2) 为 $(0, 1)$、$(1, 0)$、$(1, 1)$ 时输出 1。图 2-6 中，○表示 0，△表示 1。如果想制作或门，需要用直线将图 2-6

中的○和△分开。实际上，刚才的那条直线就将这4个点正确地分开了。

那么，换成异或门的话会如何呢？能否像或门那样，用一条直线作出分割图2-7中的○和△的空间呢？

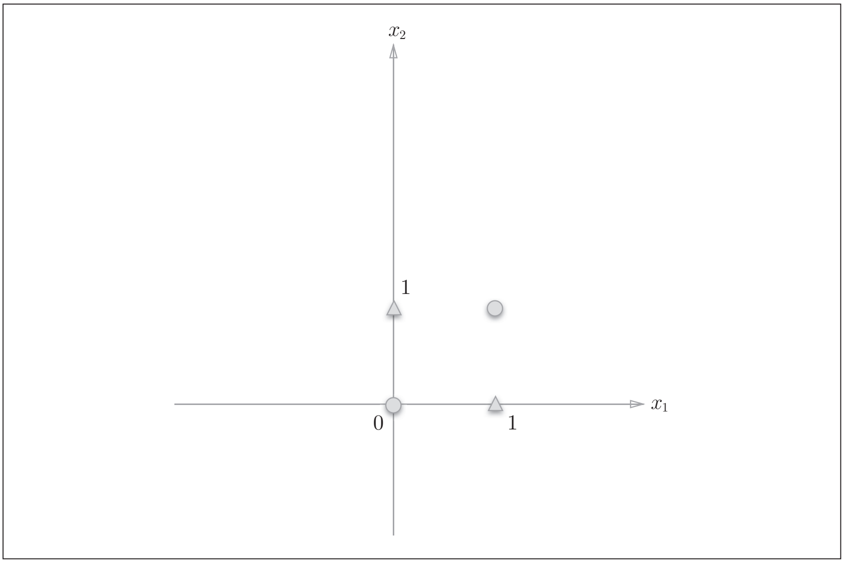


图2-7 ○和△表示异或门的输出。可否通过一条直线作出分割○和△的空间呢？

想要用一条直线将图2-7中的○和△分开，无论如何都做不到。事实上，用一条直线是无法将○和△分开的。

2.4.2 线性和非线性

图2-7中的○和△无法用一条直线分开，但是如果将"直线"这个限制条件去掉，就可以实现了。比如，我们可以像图2-8那样，作出分开○和△的空间。

感知机的局限性就在于它只能表示由一条直线分割的空间。图2-8这样弯曲的曲线无法用感知机表示。另外，由图2-8这样的曲线分割而成的空间称为**非线性**空间，由直线分割而成的空间称为**线性**空间。线性、非线性这两个术语在机器学习领域很常见，可以将其想象成图2-6和图2-8所示的直线和曲线。

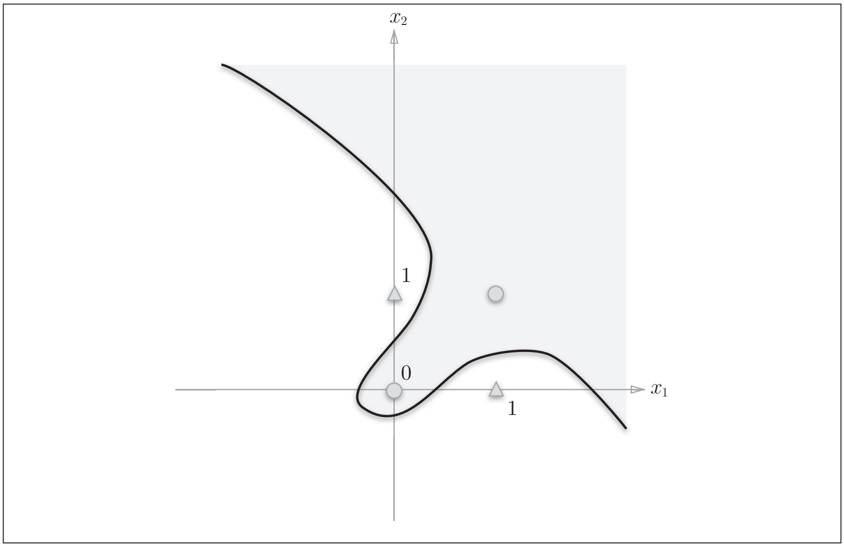


图 2-8 使用曲线可以分开○和△

2.5 多层感知机

感知机不能表示异或门让人深感遗憾，但也无需悲观。实际上，感知机的绝妙之处在于它可以“叠加层”（通过叠加层来表示异或门是本节的要点）。这里，我们暂且不考虑叠加层具体是指什么，先从其他视角来思考一下异或门的问题。

2.5.1 已有门电路的组合

异或门的制作方法有很多，其中之一就是组合我们前面做好的与门、与非门、或门进行配置。这里，与门、与非门、或门用图 2-9 中的符号表示。另外，图 2-9 中与非门前端的○表示反转输出的意思。

那么，请思考一下，要实现异或门的话，需要如何配置与门、与非门和或门呢？这里给大家一个提示，用与门、与非门、或门代替图 2-10 中的各个“？”，就可以实现异或门。

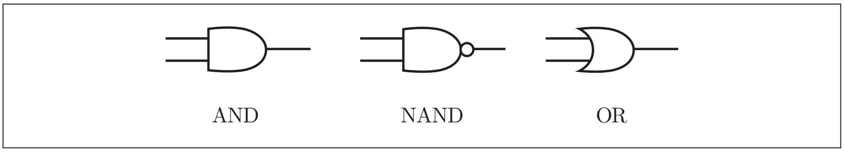


图2-9 与门、与非门、或门的符号

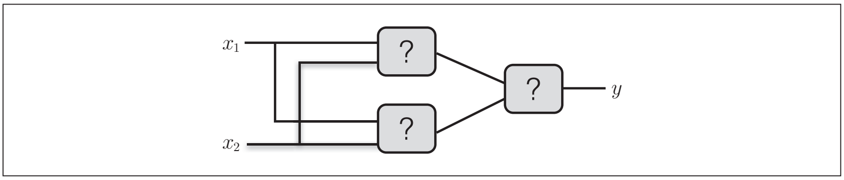


图2-10 将与门、与非门、或门代入到“？”中，就可以实现异或门！



2.4节讲到的感知机的局限性，严格地讲，应该是“单层感知机无法表示异或门”或者“单层感知机无法分离非线性空间”。接下来，我们将看到通过组合感知机（叠加层）就可以实现异或门。

异或门可以通过图2-11所示的配置来实现。这里，x_1和x_2表示输入信号，y表示输出信号。x_1和x_2是与非门和或门的输入，而与非门和或门的输出则是与门的输入。

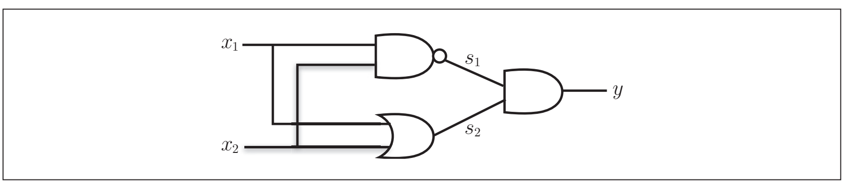


图2-11 通过组合与门、与非门、或门实现异或门

现在，我们来确认一下图2-11的配置是否真正实现了异或门。这里，把s_1作为与非门的输出，把s_2作为或门的输出，填入真值表中。结果如图2-12所示，观察x_1、x_2、y，可以发现确实符合异或门的输出。

x_1	x_2	s_1	s_2	y
0	0	1	0	0
1	0	1	1	1
0	1	1	1	1
1	1	0	1	0

图 2-12　异或门的真值表

2.5.2　异或门的实现

下面我们试着用Python来实现图2-11所示的异或门。使用之前定义的AND函数、NAND函数、OR函数，可以像下面这样（轻松地）实现。

```
def XOR(x1, x2):
    s1 = NAND(x1, x2)
    s2 = OR(x1, x2)
    y = AND(s1, s2)
    return y
```

这个XOR函数会输出预期的结果。

```
XOR(0, 0) # 输出0
XOR(1, 0) # 输出1
XOR(0, 1) # 输出1
XOR(1, 1) # 输出0
```

这样，异或门的实现就完成了。下面我们试着用感知机的表示方法（明确地显示神经元）来表示这个异或门，结果如图2-13所示。

如图2-13所示，异或门是一种多层结构的神经网络。这里，将最左边的一列称为第0层，中间的一列称为第1层，最右边的一列称为第2层。

图2-13所示的感知机与前面介绍的与门、或门的感知机（图2-1）形状不同。实际上，与门、或门是单层感知机，而异或门是2层感知机。叠加了多层的感知机也称为**多层感知机**（multi-layered perceptron）。

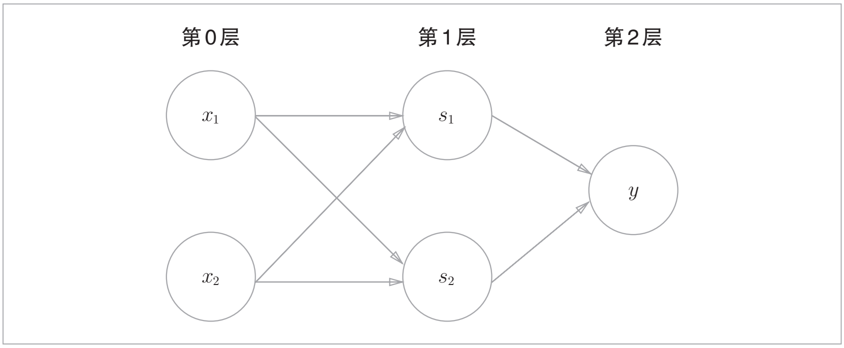


图2-13 用感知机表示异或门



图2-13中的感知机总共由3层构成，但是因为拥有权重的层实质上只有2层（第0层和第1层之间，第1层和第2层之间），所以称为“2层感知机”。不过，有的文献认为图2-13的感知机是由3层构成的，因而将其称为“3层感知机”。

在图2-13所示的2层感知机中，先在第0层和第1层的神经元之间进行信号的传送和接收，然后在第1层和第2层之间进行信号的传送和接收，具体如下所示。

1. 第0层的两个神经元接收输入信号，并将信号发送至第1层的神经元。
2. 第1层的神经元将信号发送至第2层的神经元，第2层的神经元输出y。

这种2层感知机的运行过程可以比作流水线的组装作业。第1段（第1层）的工人对传送过来的零件进行加工，完成后再传送给第2段（第2层）的工人。第2层的工人对第1层的工人传过来的零件进行加工，完成这个零件后出货（输出）。

像这样，在异或门的感知机中，工人之间不断进行零件的传送。通过这样的结构（2层结构），感知机得以实现异或门。这可以解释为“单层感知机无法表示的东西，通过增加一层就可以解决”。也就是说，通过叠加层（加深层），感知机能进行更加灵活的表示。

2.6 从与非门到计算机

多层感知机可以实现比之前见到的电路更复杂的电路。比如，进行加法运算的加法器也可以用感知机实现。此外，将二进制转换为十进制的编码器、满足某些条件就输出1的电路（用于等价检验的电路）等也可以用感知机表示。实际上，使用感知机甚至可以表示计算机！

计算机是处理信息的机器。向计算机中输入一些信息后，它会按照某种既定的方法进行处理，然后输出结果。所谓“按照某种既定的方法进行处理”是指，计算机和感知机一样，也有输入和输出，会按照某个既定的规则进行计算。

人们一般会认为计算机内部进行的处理非常复杂，而令人惊讶的是，实际上只需要通过与非门的组合，就能再现计算机进行的处理。这一令人吃惊的事实说明了什么呢？说明使用感知机也可以表示计算机。前面也介绍了，与非门可以使用感知机实现。也就是说，如果通过组合与非门可以实现计算机的话，那么通过组合感知机也可以表示计算机（感知机的组合可以通过叠加了多层的单层感知机来表示）。



说到仅通过与非门的组合就能实现计算机，大家也许一下子很难相信。建议有兴趣的读者看一下《计算机系统要素：从零开始构建现代计算机》。这本书以深入理解计算机为主题，论述了通过NAND构建可运行俄罗斯方块的计算机的过程。此书能让读者真实体会到，通过简单的NAND元件就可以实现计算机这样复杂的系统。

综上，多层感知机能够进行复杂的表示，甚至可以构建计算机。那么，什么构造的感知机才能表示计算机呢？层级多深才可以构建计算机呢？

理论上可以说2层感知机就能构建计算机。这是因为，已有研究证明，2层感知机（严格地说是激活函数使用了非线性的sigmoid函数的感知机，具体请参照下一章）可以表示任意函数。但是，使用2层感知机的构造，通过

设定合适的权重来构建计算机是一件非常累人的事情。实际上，在用与非门等低层的元件构建计算机的情况下，分阶段地制作所需的零件（模块）会比较自然，即先实现与门和或门，然后实现半加器和全加器，接着实现算数逻辑单元（ALU），然后实现CPU。因此，通过感知机表示计算机时，使用叠加了多层的构造来实现是比较自然的流程。

本书中不会实际来实现计算机，但是希望读者能够记住，感知机通过叠加层能够进行非线性的表示，理论上还可以表示计算机进行的处理。

2.7　小结

本章我们学习了感知机。感知机是一种非常简单的算法，大家应该很快就能理解它的构造。感知机是下一章要学习的神经网络的基础，因此本章的内容非常重要。

本章所学的内容

- 感知机是具有输入和输出的算法。给定一个输入后，将输出一个既定的值。
- 感知机将权重和偏置设定为参数。
- 使用感知机可以表示与门和或门等逻辑电路。
- 异或门无法通过单层感知机来表示。
- 使用2层感知机可以表示异或门。
- 单层感知机只能表示线性空间，而多层感知机可以表示非线性空间。
- 多层感知机（在理论上）可以表示计算机。

第3章
神经网络

上一章我们学习了感知机。关于感知机，既有好消息，也有坏消息。好消息是，即便对于复杂的函数，感知机也隐含着能够表示它的可能性。上一章已经介绍过，即便是计算机进行的复杂处理，感知机（理论上）也可以将其表示出来。坏消息是，设定权重的工作，即确定合适的、能符合预期的输入与输出的权重，现在还是由人工进行的。上一章中，我们结合与门、或门的真值表人工决定了合适的权重。

神经网络的出现就是为了解决刚才的坏消息。具体地讲，神经网络的一个重要性质是它可以自动地从数据中学习到合适的权重参数。本章中，我们会先介绍神经网络的概要，然后重点关注神经网络进行识别时的处理。在下一章中，我们将了解如何从数据中学习权重参数。

3.1 从感知机到神经网络

神经网络和上一章介绍的感知机有很多共同点。这里，我们主要以两者的差异为中心，来介绍神经网络的结构。

3.1.1 神经网络的例子

用图来表示神经网络的话，如图3-1所示。我们把最左边的一列称为**输入层**，最右边的一列称为**输出层**，中间的一列称为**中间层**。中间层有时

也称为**隐藏层**。“隐藏”一词的意思是，隐藏层的神经元（和输入层、输出层不同）肉眼看不见。另外，本书中把输入层到输出层依次称为第0层、第1层、第2层（层号之所以从0开始，是为了方便后面基于Python进行实现）。图3-1中，第0层对应输入层，第1层对应中间层，第2层对应输出层。

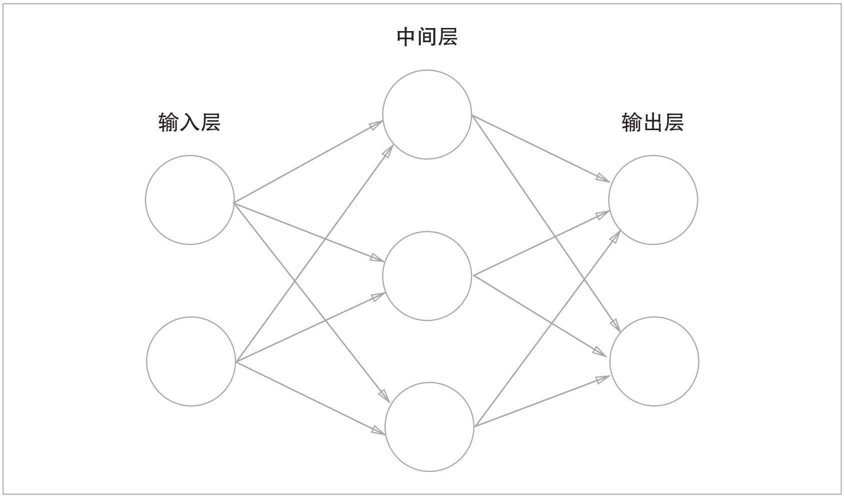


图3-1　神经网络的例子



图3-1中的网络一共由3层神经元构成，但实质上只有2层神经元有权重，因此将其称为“2层网络”。请注意，有的书也会根据构成网络的层数，把图3-1的网络称为“3层网络”。本书将根据实质上拥有权重的层数（输入层、隐藏层、输出层的总数减去1后的数量）来表示网络的名称。

只看图3-1的话，神经网络的形状类似上一章的感知机。实际上，就神经元的连接方式而言，与上一章的感知机并没有任何差异。那么，神经网络中信号是如何传递的呢？

3.1.2　复习感知机

在观察神经网络中信号的传递方法之前，我们先复习一下感知机。现在

来思考一下图3-2中的网络结构。

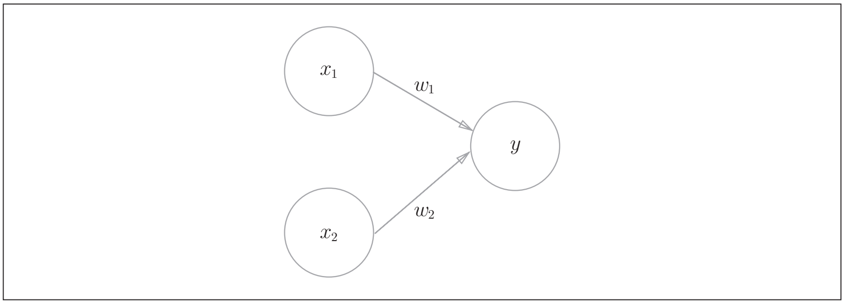


图3-2 复习感知机

图3-2中的感知机接收x_1和x_2两个输入信号，输出y。如果用数学式来表示图3-2中的感知机，则如式(3.1)所示。

$$y = \begin{cases} 0 & (b + w_1x_1 + w_2x_2 \leqslant 0) \\ 1 & (b + w_1x_1 + w_2x_2 > 0) \end{cases} \tag{3.1}$$

b是被称为偏置的参数，用于控制神经元被激活的容易程度；而w_1和w_2是表示各个信号的权重的参数，用于控制各个信号的重要性。

顺便提一下，在图3-2的网络中，偏置b并没有被画出来。如果要明确地表示出b，可以像图3-3那样做。图3-3中添加了权重为b的输入信号1。这个感知机将x_1、x_2、1三个信号作为神经元的输入，将其和各自的权重相乘后，传送至下一个神经元。在下一个神经元中，计算这些加权信号的总和。如果这个总和超过0，则输出1，否则输出0。另外，由于偏置的输入信号一直是1，所以为了区别于其他神经元，我们在图中把这个神经元整个涂成灰色。

现在将式(3.1)改写成更加简洁的形式。为了简化式(3.1)，我们用一个函数来表示这种分情况的动作(超过0则输出1，否则输出0)。引入新函数$h(x)$，将式(3.1)改写成下面的式(3.2)和式(3.3)。

$$y = h(b + w_1x_1 + w_2x_2) \tag{3.2}$$

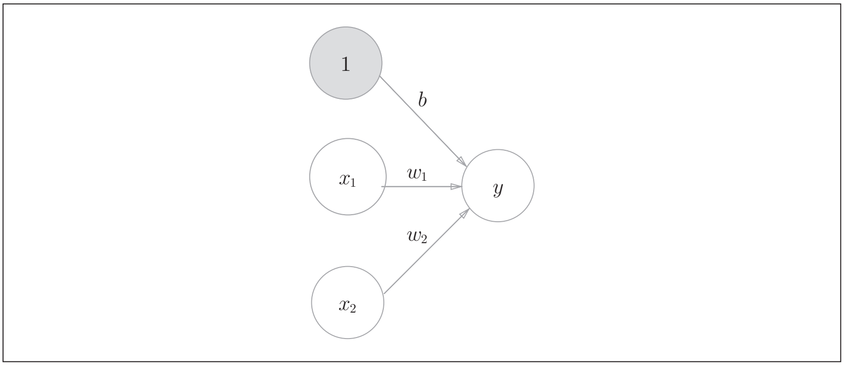


图3-3 明确表示出偏置

$$h(x) = \begin{cases} 0 & (x \leqslant 0) \\ 1 & (x > 0) \end{cases} \tag{3.3}$$

式(3.2)中，输入信号的总和会被函数$h(x)$转换，转换后的值就是输出y。然后，式(3.3)所表示的函数$h(x)$，在输入超过0时返回1，否则返回0。因此，式(3.1)和式(3.2)、式(3.3)做的是相同的事情。

3.1.3 激活函数登场

刚才登场的$h(x)$函数会将输入信号的总和转换为输出信号，这种函数一般称为**激活函数**（activation function）。如“激活”一词所示，激活函数的作用在于决定如何来激活输入信号的总和。

现在来进一步改写式(3.2)。式(3.2)分两个阶段进行处理，先计算输入信号的加权总和，然后用激活函数转换这一总和。因此，如果将式(3.2)写得详细一点，则可以分成下面两个式子。

$$a = b + w_1x_1 + w_2x_2 \tag{3.4}$$

$$y = h(a) \tag{3.5}$$

首先，式(3.4)计算加权输入信号和偏置的总和，记为a。然后，式(3.5)用$h()$函数将a转换为输出y。

之前的神经元都是用一个○表示的，如果要在图中明确表示出式(3.4)和式(3.5)，则可以像图3-4这样做。

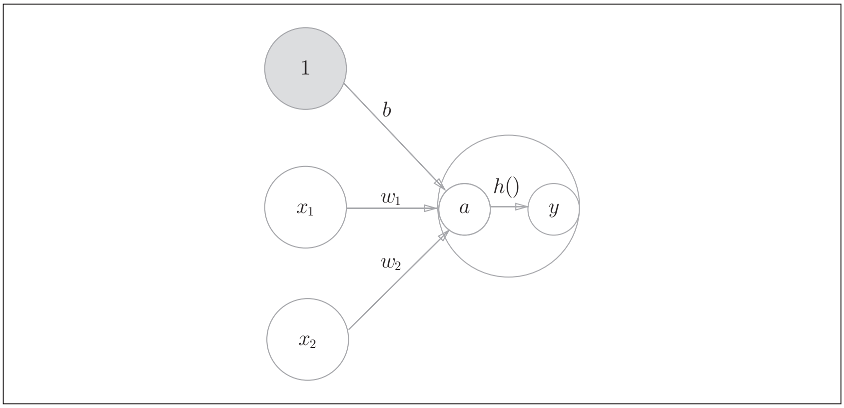


图3-4　明确显示激活函数的计算过程

如图3-4所示，表示神经元的○中明确显示了激活函数的计算过程，即信号的加权总和为节点a，然后节点a被激活函数$h()$转换成节点y。本书中，“神经元”和“节点”两个术语的含义相同。这里，我们称a和y为“节点”，其实它和之前所说的“神经元”含义相同。

通常如图3-5的左图所示，神经元用一个○表示。本书中，在可以明确神经网络的动作的情况下，将在图中明确显示激活函数的计算过程，如图3-5的右图所示。

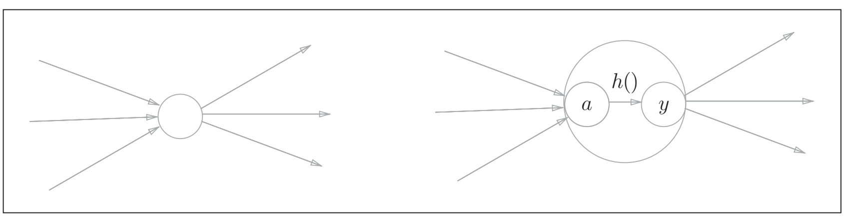


图3-5　左图是一般的神经元的图，右图是在神经元内部明确显示激活函数的计算过程的图（$\boldsymbol{a}$表示输入信号的总和，$\boldsymbol{h}()$表示激活函数，$\boldsymbol{y}$表示输出）

下面，我们将仔细介绍激活函数。激活函数是连接感知机和神经网络的桥梁。

本书在使用“感知机”一词时，没有严格统一它所指的算法。一般而言，“朴素感知机”是指单层网络，指的是激活函数使用了阶跃函数[①]的模型。“多层感知机”是指神经网络，即使用sigmoid函数（后述）等平滑的激活函数的多层网络。

3.2 激活函数

式（3.3）表示的激活函数以阈值为界，一旦输入超过阈值，就切换输出。这样的函数称为“阶跃函数”。因此，可以说感知机中使用了阶跃函数作为激活函数。也就是说，在激活函数的众多候选函数中，感知机使用了阶跃函数。那么，如果感知机使用其他函数作为激活函数的话会怎么样呢？实际上，如果将激活函数从阶跃函数换成其他函数，就可以进入神经网络的世界了。下面我们就来介绍一下神经网络使用的激活函数。

3.2.1 sigmoid函数

神经网络中经常使用的一个激活函数就是式（3.6）表示的sigmoid函数（sigmoid function）。

$$h(x) = \frac{1}{1+\exp(-x)} \tag{3.6}$$

式（3.6）中的$\exp(-x)$表示e^{-x}的意思。e是纳皮尔常数$2.7182\cdots$。式（3.6）表示的sigmoid函数看上去有些复杂，但它也仅仅是个函数而已。而函数就是给定某个输入后，会返回某个输出的转换器。比如，向sigmoid函数输入1.0或2.0后，就会有某个值被输出，类似$h(1.0) = 0.731\cdots$、$h(2.0) = 0.880\cdots$这样。

神经网络中用sigmoid函数作为激活函数，进行信号的转换，转换后的

① 阶跃函数是指一旦输入超过阈值，就切换输出的函数。

信号被传送给下一个神经元。实际上，上一章介绍的感知机和接下来要介绍的神经网络的主要区别就在于这个激活函数。其他方面，比如神经元的多层连接的构造、信号的传递方法等，基本上和感知机是一样的。下面，让我们通过和阶跃函数的比较来详细学习作为激活函数的sigmoid函数。

3.2.2 阶跃函数的实现

这里我们试着用Python画出阶跃函数的图（从视觉上确认函数的形状对理解函数而言很重要）。阶跃函数如式（3.3）所示，当输入超过0时，输出1，否则输出0。可以像下面这样简单地实现阶跃函数。

```
def step_function(x):
    if x > 0:
        return 1
    else:
        return 0
```

这个实现简单、易于理解，但是参数x只能接受实数（浮点数）。也就是说，允许形如`step_function(3.0)`的调用，但不允许参数取NumPy数组，例如`step_function(np.array([1.0, 2.0]))`。为了便于后面的操作，我们把它修改为支持NumPy数组的实现。为此，可以考虑下述实现。

```
def step_function(x):
    y = x > 0
    return y.astype(np.int)
```

上述函数的内容只有两行。由于使用了NumPy中的“技巧”，可能会有点难理解。下面我们通过Python解释器的例子来看一下这里用了什么技巧。下面这个例子中准备了NumPy数组x，并对这个NumPy数组进行了不等号运算。

```
>>> import numpy as np
>>> x = np.array([-1.0, 1.0, 2.0])
>>> x
array([-1.,  1.,  2.])
>>> y = x > 0
```

```
>>> y
array([False,  True,  True], dtype=bool)
```

对NumPy数组进行不等号运算后，数组的各个元素都会进行不等号运算，生成一个布尔型数组。这里，数组x中大于0的元素被转换为True，小于等于0的元素被转换为False，从而生成一个新的数组y。

数组y是一个布尔型数组，但是我们想要的阶跃函数是会输出int型的0或1的函数。因此，需要把数组y的元素类型从布尔型转换为int型。

```
>>> y = y.astype(np.int)
>>> y
array([0, 1, 1])
```

如上所示，可以用astype()方法转换NumPy数组的类型。astype()方法通过参数指定期望的类型，这个例子中是np.int型。Python中将布尔型转换为int型后，True会转换为1，False会转换为0。以上就是阶跃函数的实现中所用到的NumPy的“技巧”。

3.2.3 阶跃函数的图形

下面我们就用图来表示上面定义的阶跃函数，为此需要使用matplotlib库。

```
import numpy as np
import matplotlib.pylab as plt

def step_function(x):
    return np.array(x > 0, dtype=np.int)

x = np.arange(-5.0, 5.0, 0.1)
y = step_function(x)
plt.plot(x, y)
plt.ylim(-0.1, 1.1) # 指定y轴的范围
plt.show()
```

np.arange(-5.0, 5.0, 0.1)在-5.0到5.0的范围内，以0.1为单位，生成NumPy数组([-5.0, -4.9, ···, 4.9])。step_function()以该NumPy数组为参数，对数组的各个元素执行阶跃函数运算，并以数组形式返回运算结果。对数组x、y进行绘图，结果如图3-6所示。

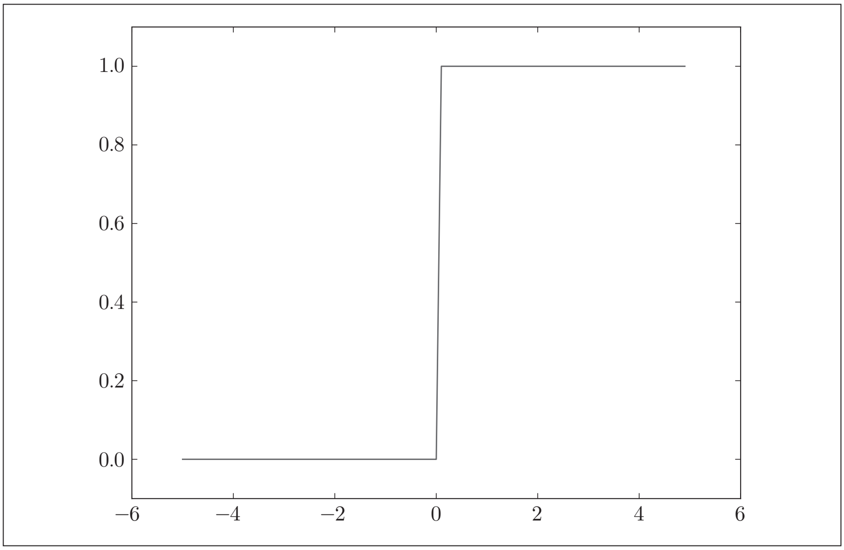


图3-6 阶跃函数的图形

如图3-6所示，阶跃函数以0为界，输出从0切换为1（或者从1切换为0）。它的值呈阶梯式变化，所以称为阶跃函数。

3.2.4 sigmoid函数的实现

下面，我们来实现sigmoid函数。用Python可以像下面这样写出式（3.6）表示的sigmoid函数。

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

这里，np.exp(-x)对应$\exp(-x)$。这个实现没有什么特别难的地方，但是要注意参数x为NumPy数组时，结果也能被正确计算。实际上，如果在这个sigmoid函数中输入一个NumPy数组，则结果如下所示。

```
>>> x = np.array([-1.0, 1.0, 2.0])
>>> sigmoid(x)
array([ 0.26894142,  0.73105858,  0.88079708])
```

之所以sigmoid函数的实现能支持NumPy数组，秘密就在于NumPy的广播功能（1.5.5节）。根据NumPy 的广播功能，如果在标量和NumPy数组之间进行运算，则标量会和NumPy数组的各个元素进行运算。这里来看一个具体的例子。

```
>>> t = np.array([1.0, 2.0, 3.0])
>>> 1.0 + t
array([ 2.,  3.,  4.])
>>> 1.0 / t
array([ 1.        ,  0.5       ,  0.33333333])
```

在这个例子中，标量（例子中是`1.0`）和NumPy数组之间进行了数值运算（+、/等）。结果，标量和NumPy数组的各个元素进行了运算，运算结果以NumPy数组的形式被输出。刚才的sigmoid函数的实现也是如此，因为`np.exp(-x)`会生成NumPy数组，所以`1 / (1 + np.exp(-x))`的运算将会在NumPy数组的各个元素间进行。

下面我们把sigmoid函数画在图上。画图的代码和刚才的阶跃函数的代码几乎是一样的，唯一不同的地方是把输出`y`的函数换成了sigmoid函数。

```
x = np.arange(-5.0, 5.0, 0.1)
y = sigmoid(x)
plt.plot(x, y)
plt.ylim(-0.1, 1.1) # 指定y轴的范围
plt.show()
```

运行上面的代码，可以得到图3-7。

3.2.5 sigmoid函数和阶跃函数的比较

现在我们来比较一下sigmoid 函数和阶跃函数，如图3-8所示。两者的不同点在哪里呢？又有哪些共同点呢？我们通过观察图3-8来思考一下。

观察图3-8，首先注意到的是“平滑性”的不同。sigmoid函数是一条平滑的曲线，输出随着输入发生连续性的变化。而阶跃函数以0为界，输出发生急剧性的变化。sigmoid函数的平滑性对神经网络的学习具有重要意义。

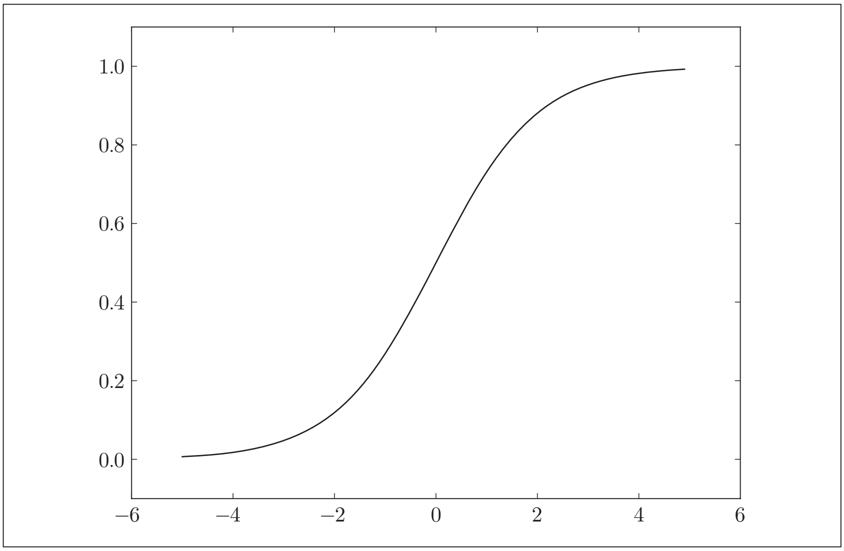


图 3-7 sigmoid 函数的图形

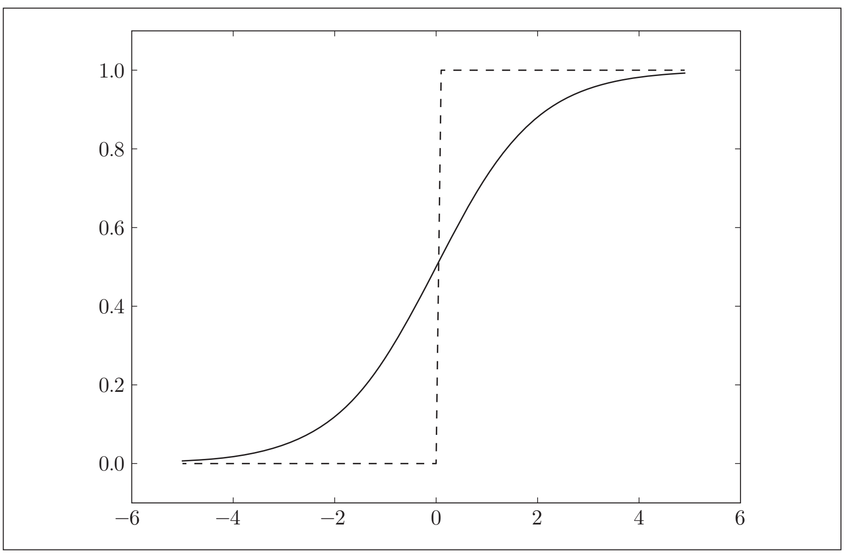


图 3-8 阶跃函数与 sigmoid 函数（虚线是阶跃函数）

另一个不同点是，相对于阶跃函数只能返回0或1，sigmoid函数可以返回0.731 ···、0.880 ···等实数(这一点和刚才的平滑性有关)。也就是说，感知机中神经元之间流动的是0或1的二元信号，而神经网络中流动的是连续的实数值信号。

如果把这两个函数与水联系起来，则阶跃函数可以比作“竹筒敲石”[1]，sigmoid函数可以比作“水车”。阶跃函数就像竹筒敲石一样，只做是否传送水(0或1)两个动作，而sigmoid函数就像水车一样，根据流过来的水量相应地调整传送出去的水量。

接着说一下阶跃函数和sigmoid函数的共同性质。阶跃函数和sigmoid函数虽然在平滑性上有差异，但是如果从宏观视角看图3-8，可以发现它们具有相似的形状。实际上，两者的结构均是“输入小时，输出接近0(为0)；随着输入增大，输出向1靠近(变成1)”。也就是说，当输入信号为重要信息时，阶跃函数和sigmoid函数都会输出较大的值；当输入信号为不重要的信息时，两者都输出较小的值。还有一个共同点是，不管输入信号有多小，或者有多大，输出信号的值都在0到1之间。

3.2.6 非线性函数

阶跃函数和sigmoid函数还有其他共同点，就是两者均为**非线性函数**。sigmoid函数是一条曲线，阶跃函数是一条像阶梯一样的折线，两者都属于非线性的函数。

在介绍激活函数时，经常会看到“非线性函数”和“线性函数”等术语。函数本来是输入某个值后会返回一个值的转换器。向这个转换器输入某个值后，输出值是输入值的常数倍的函数称为线性函数(用数学式表示为$h(x) = cx$。c为常数)。因此，线性函数是一条笔直的直线。而非线性函数，顾名思义，指的是不像线性函数那样呈现出一条直线的函数。

① 竹筒敲石是日本的一种庭院设施。支点架起竹筒，一端下方置石，另一端切口上翘。在切口上滴水，水积多后该端下垂，水流出，另一端翘起，之后又因重力而落下，击石发出响声。——译者注

神经网络的激活函数必须使用非线性函数。换句话说，激活函数不能使用线性函数。为什么不能使用线性函数呢？因为使用线性函数的话，加深神经网络的层数就没有意义了。

线性函数的问题在于，不管如何加深层数，总是存在与之等效的“无隐藏层的神经网络”。为了具体地（稍微直观地）理解这一点，我们来思考下面这个简单的例子。这里我们考虑把线性函数 $h(x) = \mathrm{c}x$ 作为激活函数，把 $y(x) = h(h(h(x)))$ 的运算对应3层神经网络[①]。这个运算会进行 $y(x) = \mathrm{c} \times \mathrm{c} \times \mathrm{c} \times x$ 的乘法运算，但是同样的处理可以由 $y(x) = ax$（注意，$a = \mathrm{c}^3$）这一次乘法运算（即没有隐藏层的神经网络）来表示。如本例所示，使用线性函数时，无法发挥多层网络带来的优势。因此，为了发挥叠加层所带来的优势，激活函数必须使用非线性函数。

3.2.7　ReLU函数

到目前为止，我们介绍了作为激活函数的阶跃函数和sigmoid函数。在神经网络发展的历史上，sigmoid函数很早就开始被使用了，而最近则主要使用**ReLU**（Rectified Linear Unit）函数。

ReLU函数在输入大于0时，直接输出该值；在输入小于等于0时，输出0（图3-9）。

ReLU函数可以表示为下面的式（3.7）。

$$h(x) = \begin{cases} x & (x > 0) \\ 0 & (x \leqslant 0) \end{cases} \tag{3.7}$$

如图3-9和式（3.7）所示，ReLU函数是一个非常简单的函数。因此，ReLU函数的实现也很简单，可以写成如下形式。

```
def relu(x):
    return np.maximum(0, x)
```

① 该对应只是一个近似，实际的神经网络运算比这个例子要复杂，但不影响后面的结论成立。
——译者注

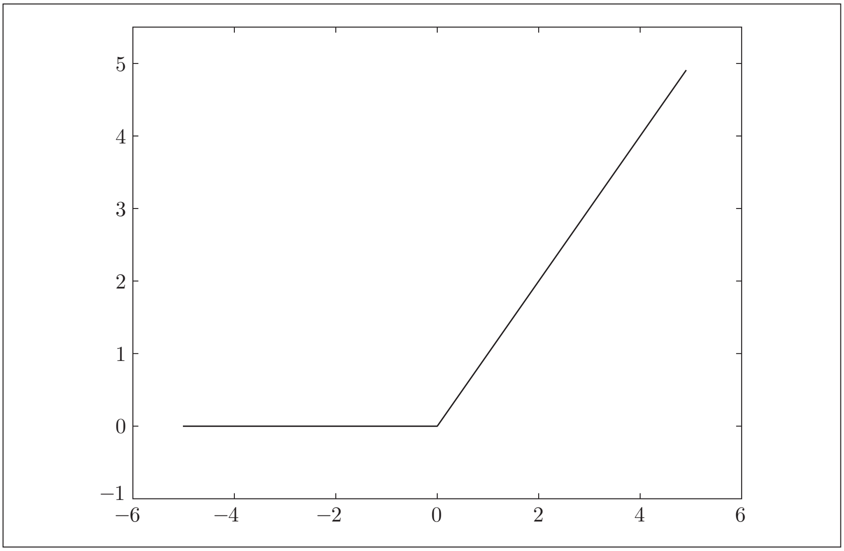


图3-9　ReLU函数

这里使用了NumPy的`maximum`函数。`maximum`函数会从输入的数值中选择较大的那个值进行输出。

本章剩余部分的内容仍将使用sigmoid函数作为激活函数，但在本书的后半部分，则将主要使用ReLU函数。

3.3　多维数组的运算

如果掌握了NumPy多维数组的运算，就可以高效地实现神经网络。因此，本节将介绍NumPy多维数组的运算，然后再进行神经网络的实现。

3.3.1　多维数组

简单地讲，多维数组就是“数字的集合”，数字排成一列的集合、排成长方形的集合、排成三维状或者（更加一般化的）N维状的集合都称为多维数组。下面我们就用NumPy来生成多维数组，先从前面介绍过的一维数组开始。

```
>>> import numpy as np
>>> A = np.array([1, 2, 3, 4])
>>> print(A)
[1 2 3 4]
>>> np.ndim(A)
1
>>> A.shape
(4,)
>>> A.shape[0]
4
```

如上所示，数组的维数可以通过np.dim()函数获得。此外，数组的形状可以通过实例变量shape获得。在上面的例子中，A是一维数组，由4个元素构成。注意，这里的A.shape的结果是个元组（tuple）。这是因为一维数组的情况下也要返回和多维数组的情况下一致的结果。例如，二维数组时返回的是元组(4,3)，三维数组时返回的是元组(4,3,2)，因此一维数组时也同样以元组的形式返回结果。下面我们来生成一个二维数组。

```
>>> B = np.array([[1,2], [3,4], [5,6]])
>>> print(B)
[[1 2]
 [3 4]
 [5 6]]
>>> np.ndim(B)
2
>>> B.shape
(3, 2)
```

这里生成了一个3×2的数组B。3×2的数组表示第一个维度有3个元素，第二个维度有2个元素。另外，第一个维度对应第0维，第二个维度对应第1维（Python的索引从0开始）。二维数组也称为**矩阵**（matrix）。如图3-10所示，数组的横向排列称为**行**（row），纵向排列称为**列**（column）。

3.3.2　矩阵乘法

下面，我们来介绍矩阵（二维数组）的乘积。比如2×2的矩阵，其乘积可以像图3-11这样进行计算（按图中顺序进行计算是规定好了的）。

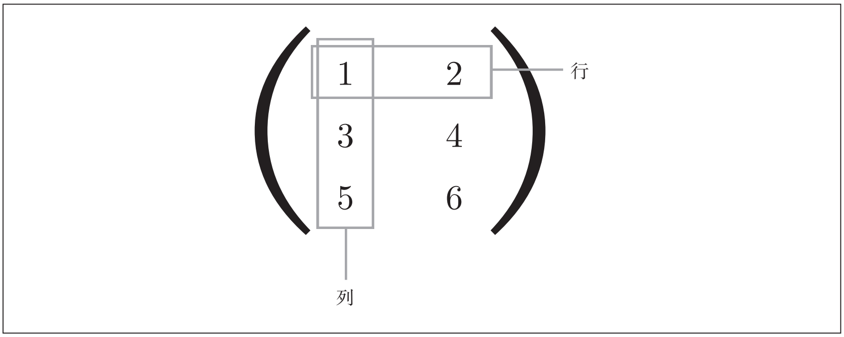


图3-10 横向排列称为行，纵向排列称为列

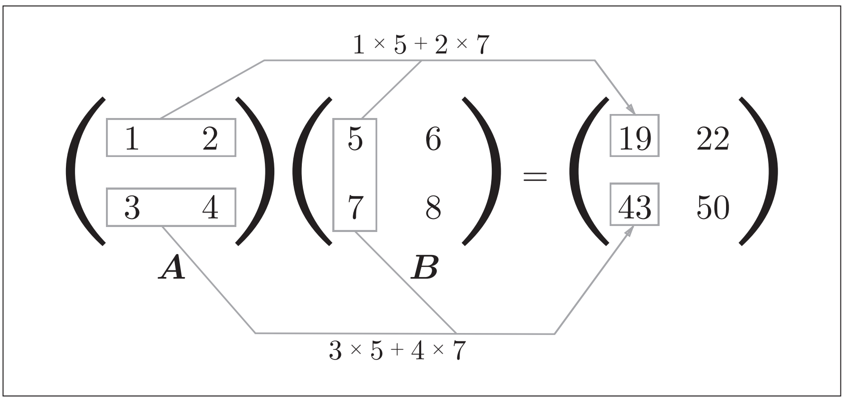


图3-11 矩阵的乘积的计算方法

如本例所示，矩阵的乘积是通过左边矩阵的行（横向）和右边矩阵的列（纵向）以对应元素的方式相乘后再求和而得到的。并且，运算的结果保存为新的多维数组的元素。比如，$\boldsymbol{A}$ 的第1行和 $\boldsymbol{B}$ 的第1列的乘积结果是新数组的第1行第1列的元素，$\boldsymbol{A}$ 的第2行和 $\boldsymbol{B}$ 的第1列的结果是新数组的第2行第1列的元素。另外，在本书的数学标记中，矩阵将用黑斜体表示（比如，矩阵 $\boldsymbol{A}$），以区别于单个元素的标量（比如，a 或 b）。这个运算在Python中可以用如下代码实现。

```
>>> A = np.array([[1,2], [3,4]])
```

```
>>> A.shape
(2, 2)
>>> B = np.array([[5,6], [7,8]])
>>> B.shape
(2, 2)
>>> np.dot(A, B)
array([[19, 22],
       [43, 50]])
```

这里，$\boldsymbol{A}$和$\boldsymbol{B}$都是2×2的矩阵，它们的乘积可以通过NumPy的np.dot()函数计算（乘积也称为点积）。np.dot()接收两个NumPy数组作为参数，并返回数组的乘积。这里要注意的是，np.dot(A, B)和np.dot(B, A)的值可能不一样。和一般的运算（+或*等）不同，矩阵的乘积运算中，操作数（A、B）的顺序不同，结果也会不同。

这里介绍的是计算2×2形状的矩阵的乘积的例子，其他形状的矩阵的乘积也可以用相同的方法来计算。比如，2×3的矩阵和3×2 的矩阵的乘积可按如下形式用Python来实现。

```
>>> A = np.array([[1,2,3], [4,5,6]])
>>> A.shape
(2, 3)
>>> B = np.array([[1,2], [3,4], [5,6]])
>>> B.shape
(3, 2)
>>> np.dot(A, B)
array([[22, 28],
       [49, 64]])
```

2×3的矩阵$\boldsymbol{A}$和3×2的矩阵$\boldsymbol{B}$的乘积可按以上方式实现。这里需要注意的是矩阵的形状（shape）。具体地讲，矩阵$\boldsymbol{A}$的第1维的元素个数（列数）必须和矩阵$\boldsymbol{B}$的第0维的元素个数（行数）相等。在上面的例子中，矩阵$\boldsymbol{A}$的形状是2×3，矩阵$\boldsymbol{B}$的形状是3×2，矩阵$\boldsymbol{A}$的第1维的元素个数（3）和矩阵$\boldsymbol{B}$的第0维的元素个数（3）相等。如果这两个值不相等，则无法计算矩阵的乘积。比如，如果用Python计算2×3 的矩阵$\boldsymbol{A}$和2×2的矩阵$\boldsymbol{C}$的乘积，则会输出如下错误。

```
>>> C = np.array([[1,2], [3,4]])
>>> C.shape
```

```
(2, 2)
>>> A.shape
(2, 3)
>>> np.dot(A, C)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: shapes (2,3) and (2,2) not aligned: 3 (dim 1) != 2 (dim 0)
```

这个错误的意思是，矩阵 $\boldsymbol{A}$ 的第1维和矩阵 $\boldsymbol{C}$ 的第0维的元素个数不一致（维度的索引从0开始）。也就是说，在多维数组的乘积运算中，必须使两个矩阵中的对应维度的元素个数一致，这一点很重要。我们通过图3-12再来确认一下。

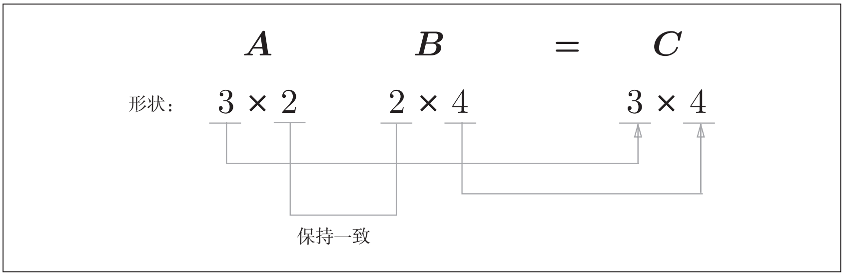


图3-12　在矩阵的乘积运算中，对应维度的元素个数要保持一致

图3-12中，3×2 的矩阵 $\boldsymbol{A}$ 和 2×4 的矩阵 $\boldsymbol{B}$ 的乘积运算生成了 3×4 的矩阵 $\boldsymbol{C}$。如图所示，矩阵 $\boldsymbol{A}$ 和矩阵 $\boldsymbol{B}$ 的对应维度的元素个数必须保持一致。此外，还有一点很重要，就是运算结果的矩阵 $\boldsymbol{C}$ 的形状是由矩阵 $\boldsymbol{A}$ 的行数和矩阵 $\boldsymbol{B}$ 的列数构成的。

另外，当 $\boldsymbol{A}$ 是二维矩阵、$\boldsymbol{B}$ 是一维数组时，如图3-13所示，对应维度的元素个数要保持一致的原则依然成立。

可按如下方式用Python实现图3-13的例子。

```
>>> A = np.array([[1,2], [3, 4], [5,6]])
>>> A.shape
(3, 2)
>>> B = np.array([7,8])
>>> B.shape
(2,)
>>> np.dot(A, B)
array([23, 53, 83])
```

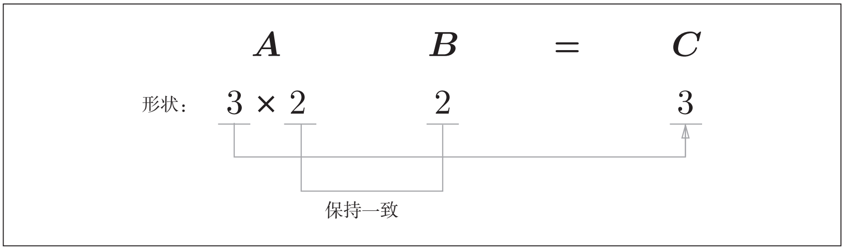


图3-13 $\boldsymbol{A}$是二维矩阵、$\boldsymbol{B}$是一维数组时，也要保持对应维度的元素个数一致

3.3.3 神经网络的内积

下面我们使用NumPy矩阵来实现神经网络。这里我们以图3-14中的简单神经网络为对象。这个神经网络省略了偏置和激活函数，只有权重。

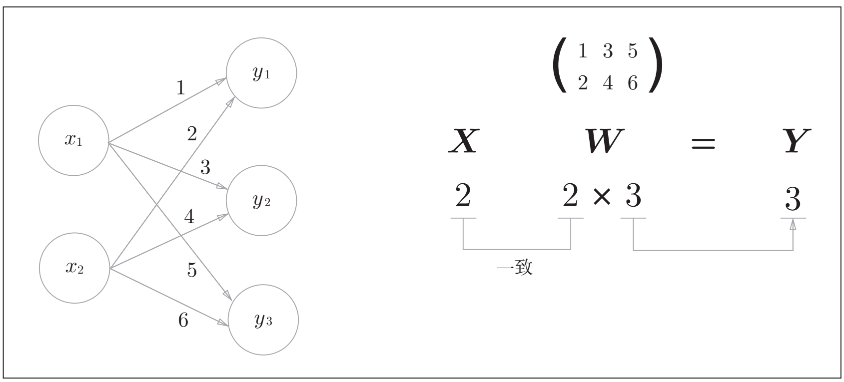


图3-14 通过矩阵的乘积进行神经网络的运算

实现该神经网络时，要注意$\boldsymbol{X}$、$\boldsymbol{W}$、$\boldsymbol{Y}$的形状，特别是$\boldsymbol{X}$和$\boldsymbol{W}$的对应维度的元素个数是否一致，这一点很重要。

```
>>> X = np.array([1, 2])
>>> X.shape
(2,)
>>> W = np.array([[1, 3, 5], [2, 4, 6]])
>>> print(W)
[[1 3 5]
 [2 4 6]]
>>> W.shape
```

```
(2, 3)
>>> Y = np.dot(X, W)
>>> print(Y)
[ 5  11  17]
```

如上所示，使用np.dot(多维数组的点积)，可以一次性计算出$\boldsymbol{Y}$的结果。这意味着，即便$\boldsymbol{Y}$的元素个数为100或1000，也可以通过一次运算就计算出结果！如果不使用np.dot，就必须单独计算$\boldsymbol{Y}$的每一个元素(或者说必须使用for语句)，非常麻烦。因此，通过矩阵的乘积一次性完成计算的技巧，在实现的层面上可以说是非常重要的。

3.4 3层神经网络的实现

现在我们来进行神经网络的实现。这里我们以图3-15的3层神经网络为对象，实现从输入到输出的(前向)处理。在代码实现方面，使用上一节介绍的NumPy多维数组。巧妙地使用NumPy数组，可以用很少的代码完成神经网络的前向处理。

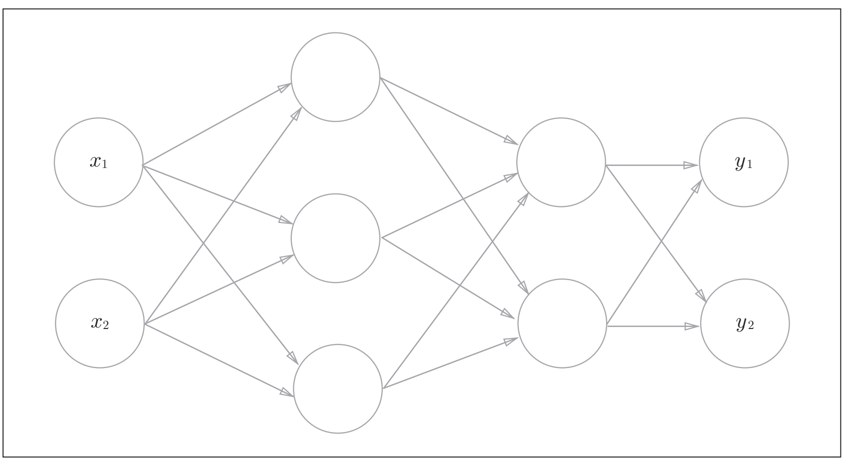


图3-15 3层神经网络：输入层(第0层)有2个神经元，第1个隐藏层(第1层)有3个神经元，第2个隐藏层(第2层)有2个神经元，输出层(第3层)有2个神经元

3.4.1　符号确认

在介绍神经网络中的处理之前，我们先导入 $w_{12}^{(1)}$、$a_1^{(1)}$ 等符号。这些符号可能看上去有些复杂，不过因为只在本节使用，稍微读一下就跳过去也问题不大。



本节的重点是神经网络的运算可以作为矩阵运算打包进行。因为神经网络各层的运算是通过矩阵的乘法运算打包进行的（从宏观视角来考虑），所以即便忘了（未记忆）具体的符号规则，也不影响理解后面的内容。

我们先从定义符号开始。请看图3-16。图3-16中只突出显示了从输入层神经元 x_2 到后一层的神经元 $a_1^{(1)}$ 的权重。

如图3-16所示，权重和隐藏层的神经元的右上角有一个“(1)”，它表示权重和神经元的层号（即第1层的权重、第1层的神经元）。此外，权重的右下角有两个数字，它们是后一层的神经元和前一层的神经元的索引号。比如，$w_{12}^{(1)}$ 表示前一层的第2个神经元 x_2 到后一层的第1个神经元 $a_1^{(1)}$ 的权重。权重右下角按照“后一层的索引号、前一层的索引号”的顺序排列。

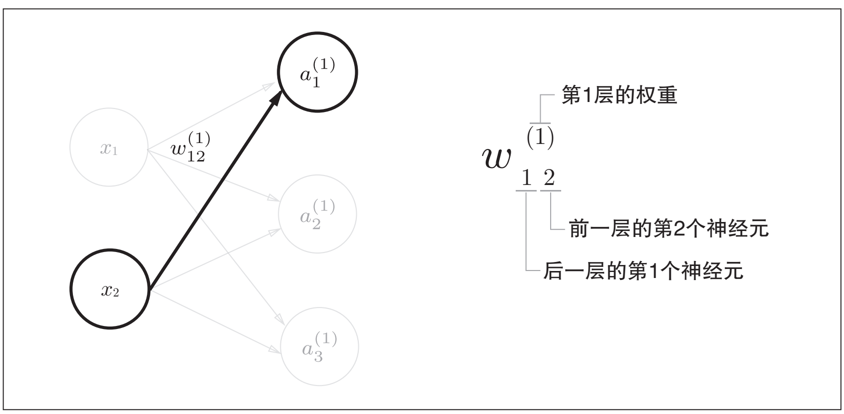


图3-16　权重的符号

3.4.2 各层间信号传递的实现

现在看一下从输入层到第1层的第1个神经元的信号传递过程，如图3-17所示。

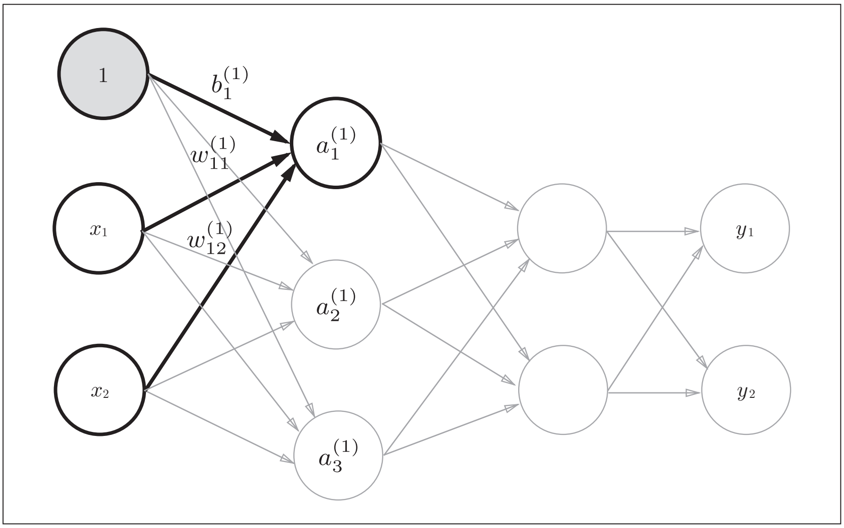


图3-17 从输入层到第1层的信号传递

图3-17中增加了表示偏置的神经元“1”。请注意，偏置的右下角的索引号只有一个。这是因为前一层的偏置神经元（神经元“1”）只有一个[①]。

为了确认前面的内容，现在用数学式表示 $a_1^{(1)}$。$a_1^{(1)}$ 通过加权信号和偏置的和按如下方式进行计算。

$$a_1^{(1)} = w_{11}^{(1)} x_1 + w_{12}^{(1)} x_2 + b_1^{(1)} \tag{3.8}$$

① 任何前一层的偏置神经元“1”都只有一个。偏置权重的数量取决于后一层的神经元的数量（不包括后一层的偏置神经元“1”）。——译者注

此外，如果使用矩阵的乘法运算，则可以将第1层的加权和表示成下面的式(3.9)。

$$A^{(1)} = XW^{(1)} + B^{(1)} \tag{3.9}$$

其中，$A^{(1)}$、X、$B^{(1)}$、$W^{(1)}$如下所示。

$$A^{(1)} = \begin{pmatrix} a_1^{(1)} & a_2^{(1)} & a_3^{(1)} \end{pmatrix}，X = \begin{pmatrix} x_1 & x_2 \end{pmatrix}，B^{(1)} = \begin{pmatrix} b_1^{(1)} & b_2^{(1)} & b_3^{(1)} \end{pmatrix}$$

$$W^{(1)} = \begin{pmatrix} w_{11}^{(1)} & w_{21}^{(1)} & w_{31}^{(1)} \\ w_{12}^{(1)} & w_{22}^{(1)} & w_{32}^{(1)} \end{pmatrix}$$

下面我们用NumPy多维数组来实现式(3.9)，这里将输入信号、权重、偏置设置成任意值。

```
X = np.array([1.0, 0.5])
W1 = np.array([[0.1, 0.3, 0.5], [0.2, 0.4, 0.6]])
B1 = np.array([0.1, 0.2, 0.3])

print(W1.shape) # (2, 3)
print(X.shape)  # (2,)
print(B1.shape) # (3,)

A1 = np.dot(X, W1) + B1
```

这个运算和上一节进行的运算是一样的。`W1`是2×3的数组，`X`是元素个数为2的一维数组。这里，`W1`和`X`的对应维度的元素个数也保持了一致。

接下来，我们观察第1层中激活函数的计算过程。如果把这个计算过程用图来表示的话，则如图3-18所示。

如图3-18所示，隐藏层的加权和(加权信号和偏置的总和)用a表示，被激活函数转换后的信号用z表示。此外，图中$h()$表示激活函数，这里我们使用的是sigmoid函数。用Python来实现，代码如下所示。

```
Z1 = sigmoid(A1)

print(A1) # [0.3, 0.7, 1.1]
print(Z1) # [0.57444252, 0.66818777, 0.75026011]
```

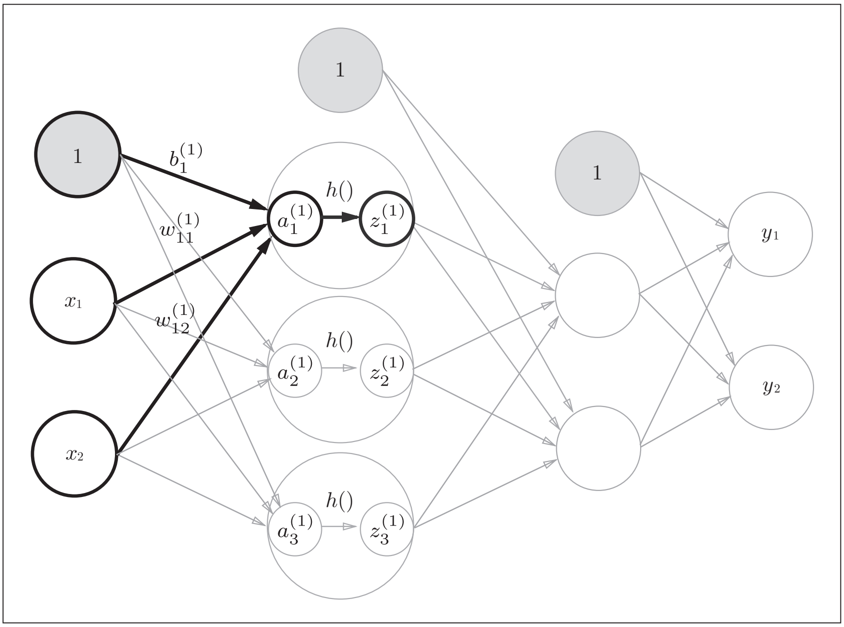


图3-18　从输入层到第1层的信号传递

这个`sigmoid()`函数就是之前定义的那个函数。它会接收NumPy数组，并返回元素个数相同的NumPy数组。

下面，我们来实现第1层到第2层的信号传递（图3-19）。

```
W2 = np.array([[0.1, 0.4], [0.2, 0.5], [0.3, 0.6]])
B2 = np.array([0.1, 0.2])

print(Z1.shape) # (3,)
print(W2.shape) # (3, 2)
print(B2.shape) # (2,)

A2 = np.dot(Z1, W2) + B2
Z2 = sigmoid(A2)
```

除了第1层的输出（`Z1`）变成了第2层的输入这一点以外，这个实现和刚才的代码完全相同。由此可知，通过使用NumPy数组，可以将层到层的信号传递过程简单地写出来。

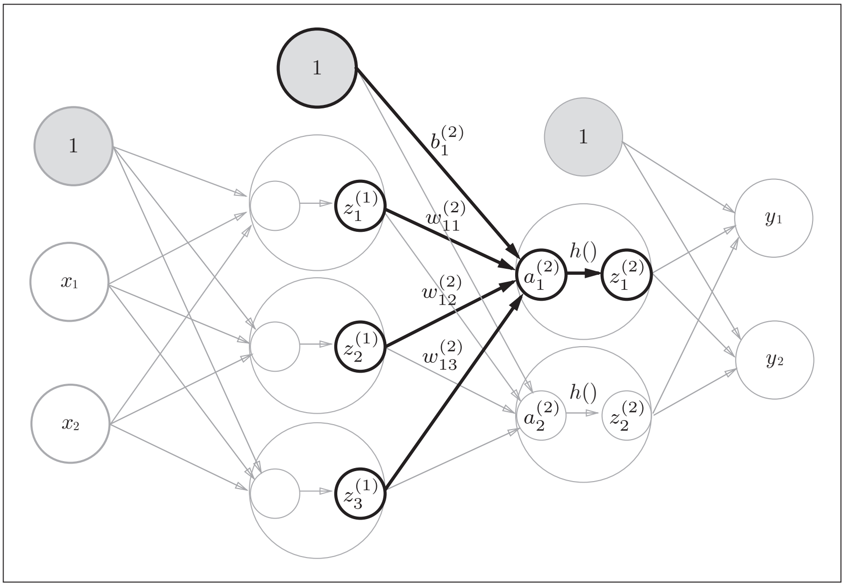


图3-19　第1层到第2层的信号传递

最后是第2层到输出层的信号传递（图3-20）。输出层的实现也和之前的实现基本相同。不过，最后的激活函数和之前的隐藏层有所不同。

```
def identity_function(x):
    return x

W3 = np.array([[0.1, 0.3], [0.2, 0.4]])
B3 = np.array([0.1, 0.2])

A3 = np.dot(Z2, W3) + B3
Y = identity_function(A3) # 或者Y = A3
```

这里我们定义了identity_function()函数（也称为"恒等函数"），并将其作为输出层的激活函数。恒等函数会将输入按原样输出，因此，这个例子中没有必要特意定义identity_function()。这里这样实现只是为了和之前的流程保持统一。另外，图3-20中，输出层的激活函数用$\sigma()$表示，不同于隐藏层的激活函数$h()$（σ读作sigma）。

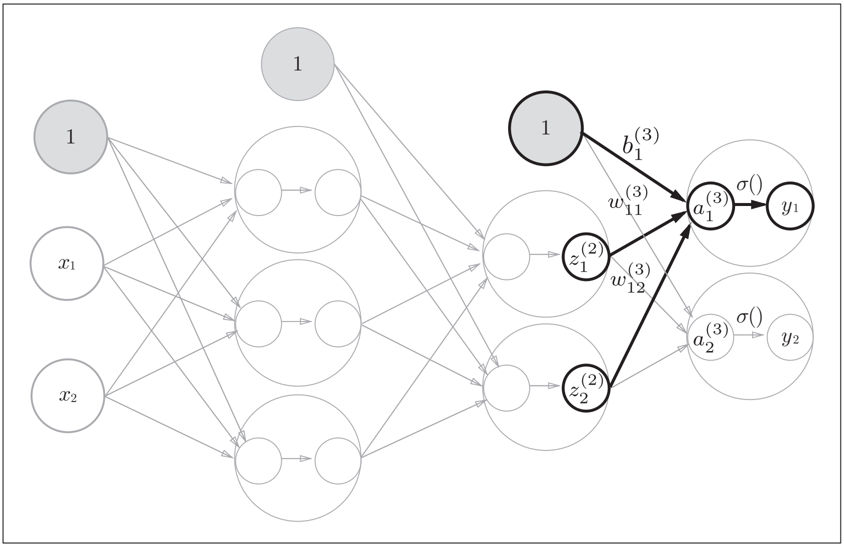


图3-20　从第2层到输出层的信号传递

> 输出层所用的激活函数，要根据求解问题的性质决定。一般地，回归问题可以使用恒等函数，二元分类问题可以使用sigmoid函数，多元分类问题可以使用softmax函数。关于输出层的激活函数，我们将在下一节详细介绍。

3.4.3　代码实现小结

至此，我们已经介绍完了3层神经网络的实现。现在我们把之前的代码实现全部整理一下。这里，我们按照神经网络的实现惯例，只把权重记为大写字母`W1`，其他的（偏置或中间结果等）都用小写字母表示。

```
def init_network():
    network = {}
    network['W1'] = np.array([[0.1, 0.3, 0.5], [0.2, 0.4, 0.6]])
    network['b1'] = np.array([0.1, 0.2, 0.3])
    network['W2'] = np.array([[0.1, 0.4], [0.2, 0.5], [0.3, 0.6]])
    network['b2'] = np.array([0.1, 0.2])
    network['W3'] = np.array([[0.1, 0.3], [0.2, 0.4]])
```

```
    network['b3'] = np.array([0.1, 0.2])

    return network

def forward(network, x):
    W1, W2, W3 = network['W1'], network['W2'], network['W3']
    b1, b2, b3 = network['b1'], network['b2'], network['b3']

    a1 = np.dot(x, W1) + b1
    z1 = sigmoid(a1)
    a2 = np.dot(z1, W2) + b2
    z2 = sigmoid(a2)
    a3 = np.dot(z2, W3) + b3
    y = identity_function(a3)

    return y

network = init_network()
x = np.array([1.0, 0.5])
y = forward(network, x)
print(y) # [ 0.31682708  0.69627909]
```

这里定义了 `init_network()` 和 `forward()` 函数。`init_network()` 函数会进行权重和偏置的初始化，并将它们保存在字典变量 `network` 中。这个字典变量 `network` 中保存了每一层所需的参数（权重和偏置）。`forward()` 函数中则封装了将输入信号转换为输出信号的处理过程。

另外，这里出现了 forward（前向）一词，它表示的是从输入到输出方向的传递处理。后面在进行神经网络的训练时，我们将介绍后向（backward，从输出到输入方向）的处理。

至此，神经网络的前向处理的实现就完成了。通过巧妙地使用 NumPy 多维数组，我们高效地实现了神经网络。

3.5 输出层的设计

神经网络可以用在分类问题和回归问题上，不过需要根据情况改变输出层的激活函数。一般而言，回归问题用恒等函数，分类问题用 softmax 函数。

机器学习的问题大致可以分为分类问题和回归问题。分类问题是数据属于哪一个类别的问题。比如，区分图像中的人是男性还是女性的问题就是分类问题。而回归问题是根据某个输入预测一个（连续的）数值的问题。比如，根据一个人的图像预测这个人的体重的问题就是回归问题（类似“57.4kg”这样的预测）。

3.5.1 恒等函数和softmax函数

恒等函数会将输入按原样输出，对于输入的信息，不加以任何改动地直接输出。因此，在输出层使用恒等函数时，输入信号会原封不动地被输出。另外，将恒等函数的处理过程用之前的神经网络图来表示的话，则如图3-21所示。和前面介绍的隐藏层的激活函数一样，恒等函数进行的转换处理可以用一根箭头来表示。

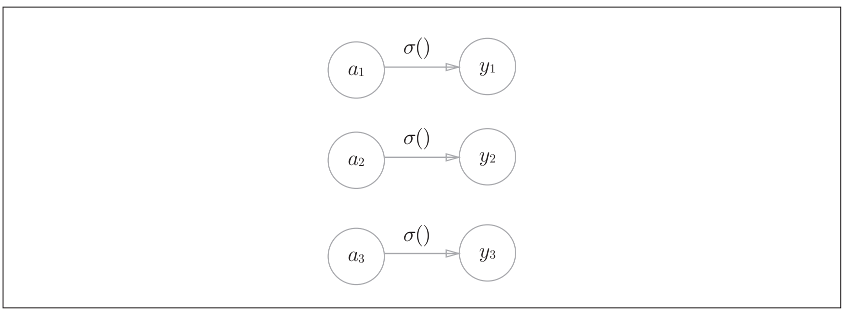


图3-21 恒等函数

分类问题中使用的softmax函数可以用下面的式（3.10）表示。

$$y_k = \frac{\exp(a_k)}{\sum\limits_{i=1}^{n} \exp(a_i)} \tag{3.10}$$

$\exp(x)$是表示e^x的指数函数（e是纳皮尔常数2.7182…）。式（3.10）表示假设输出层共有n个神经元，计算第k个神经元的输出y_k。如式（3.10）所示，softmax函数的分子是输入信号a_k的指数函数，分母是所有输入信号的指数函数的和。

用图表示softmax函数的话，如图3-22所示。图3-22中，softmax函数的输出通过箭头与所有的输入信号相连。这是因为，从式(3.10)可以看出，输出层的各个神经元都受到所有输入信号的影响。

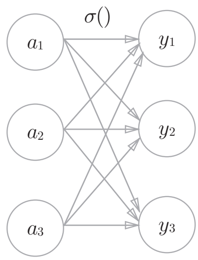


图3-22　softmax函数

现在我们来实现softmax函数。在这个过程中，我们将使用Python解释器逐一确认结果。

```
>>> a = np.array([0.3, 2.9, 4.0])
>>>
>>> exp_a = np.exp(a) # 指数函数
>>> print(exp_a)
[  1.34985881  18.17414537  54.59815003]
>>>
>>> sum_exp_a = np.sum(exp_a) # 指数函数的和
>>> print(sum_exp_a)
74.1221542102
>>>
>>> y = exp_a / sum_exp_a
>>> print(y)
[ 0.01821127  0.24519181  0.73659691]
```

这个Python实现是完全依照式(3.10)进行的，所以不需要特别的解释。考虑到后面还要使用softmax函数，这里我们把它定义成如下的Python函数。

```
def softmax(a):
    exp_a = np.exp(a)
    sum_exp_a = np.sum(exp_a)
    y = exp_a / sum_exp_a

    return y
```

3.5.2 实现softmax函数时的注意事项

上面的`softmax`函数的实现虽然正确描述了式(3.10)，但在计算机的运算上有一定的缺陷。这个缺陷就是溢出问题。softmax函数的实现中要进行指数函数的运算，但是此时指数函数的值很容易变得非常大。比如，e^{10}的值会超过20000，e^{100}会变成一个后面有40多个0的超大值，e^{1000}的结果会返回一个表示无穷大的`inf`。如果在这些超大值之间进行除法运算，结果会出现“不确定”的情况。



计算机处理“数”时，数值必须在4字节或8字节的有限数据宽度内。这意味着数存在有效位数，也就是说，可以表示的数值范围是有限的。因此，会出现超大值无法表示的问题。这个问题称为溢出，在进行计算机的运算时必须(常常)注意。

softmax函数的实现可以像式(3.11)这样进行改进。

$$
\begin{aligned}
y_k = \frac{\exp(a_k)}{\sum\limits_{i=1}^{n}\exp(a_i)} &= \frac{\mathrm{C}\exp(a_k)}{\mathrm{C}\sum\limits_{i=1}^{n}\exp(a_i)} \\
&= \frac{\exp(a_k + \log \mathrm{C})}{\sum\limits_{i=1}^{n}\exp(a_i + \log \mathrm{C})} \\
&= \frac{\exp(a_k + \mathrm{C}')}{\sum\limits_{i=1}^{n}\exp(a_i + \mathrm{C}')}
\end{aligned}
\tag{3.11}
$$

首先，式(3.11)在分子和分母上都乘上C这个任意的常数(因为同时对分母和分子乘以相同的常数，所以计算结果不变)。然后，把这个C移动到指数函数(exp)中，记为$\log \mathrm{C}$。最后，把$\log \mathrm{C}$替换为另一个符号C'。

式(3.11)说明，在进行softmax的指数函数的运算时，加上(或者减去)某个常数并不会改变运算的结果。这里的C'可以使用任何值，但是为了防止溢出，一般会使用输入信号中的最大值。我们来看一个具体的例子。

```
>>> a = np.array([1010, 1000, 990])
>>> np.exp(a) / np.sum(np.exp(a)) # softmax函数的运算
array([ nan,  nan,  nan])         # 没有被正确计算
>>>
>>> c = np.max(a) # 1010
>>> a - c
array([  0, -10, -20])
>>>
>>> np.exp(a - c) / np.sum(np.exp(a - c))
array([  9.99954600e-01,   4.53978686e-05,   2.06106005e-09])
```

如该例所示，通过减去输入信号中的最大值（上例中的c），我们发现原本为nan（not a number，不确定）的地方，现在被正确计算了。综上，我们可以像下面这样实现softmax函数。

```
def softmax(a):
    c = np.max(a)
    exp_a = np.exp(a - c) # 溢出对策
    sum_exp_a = np.sum(exp_a)
    y = exp_a / sum_exp_a

    return y
```

3.5.3 softmax函数的特征

使用softmax()函数，可以按如下方式计算神经网络的输出。

```
>>> a = np.array([0.3, 2.9, 4.0])
>>> y = softmax(a)
>>> print(y)
[ 0.01821127  0.24519181  0.73659691]
>>> np.sum(y)
1.0
```

如上所示，softmax函数的输出是0.0到1.0之间的实数。并且，softmax函数的输出值的总和是1。输出总和为1是softmax函数的一个重要性质。正因为有了这个性质，我们才可以把softmax函数的输出解释为“概率”。

比如，上面的例子可以解释成y[0]的概率是0.018（1.8％），y[1]的概率是0.245（24.5％），y[2]的概率是0.737（73.7％）。从概率的结果来看，可以说“因为第2个元素的概率最高，所以答案是第2个类别”。而且，还可以回

答“有74 %的概率是第2个类别，有25 %的概率是第1个类别，有1 %的概率是第0个类别”。也就是说，通过使用softmax函数，我们可以用概率的（统计的）方法处理问题。

这里需要注意的是，即便使用了softmax函数，各个元素之间的大小关系也不会改变。这是因为指数函数（$y = \exp(x)$）是单调递增函数。实际上，上例中a的各元素的大小关系和y的各元素的大小关系并没有改变。比如，a的最大值是第2个元素，y的最大值也仍是第2个元素。

一般而言，神经网络只把输出值最大的神经元所对应的类别作为识别结果。并且，即便使用softmax函数，输出值最大的神经元的位置也不会变。因此，神经网络在进行分类时，输出层的softmax函数可以省略。在实际的问题中，由于指数函数的运算需要一定的计算机运算量，因此输出层的softmax函数一般会被省略。



求解机器学习问题的步骤可以分为“学习”[①]和“推理”两个阶段。首先，在学习阶段进行模型的学习[②]，然后，在推理阶段，用学到的模型对未知的数据进行推理（分类）。如前所述，推理阶段一般会省略输出层的softmax函数。在输出层使用softmax函数是因为它和神经网络的学习有关系（详细内容请参考下一章）。

3.5.4 输出层的神经元数量

输出层的神经元数量需要根据待解决的问题来决定。对于分类问题，输出层的神经元数量一般设定为类别的数量。比如，对于某个输入图像，预测是图中的数字0到9中的哪一个的问题（10类别分类问题），可以像图3-23这样，将输出层的神经元设定为10个。

如图3-23所示，在这个例子中，输出层的神经元从上往下依次对应数字$0, 1, \cdots, 9$。此外，图中输出层的神经元的值用不同的灰度表示。这个例子

① “学习”也称为“训练”，为了强调算法从数据中学习模型，本书使用“学习”一词。——译者注

② 这里的“学习”是指使用训练数据、自动调整参数的过程，具体请参考第4章。——译者注

中神经元 y_2 颜色最深，输出的值最大。这表明这个神经网络预测的是 y_2 对应的类别，也就是“2”。

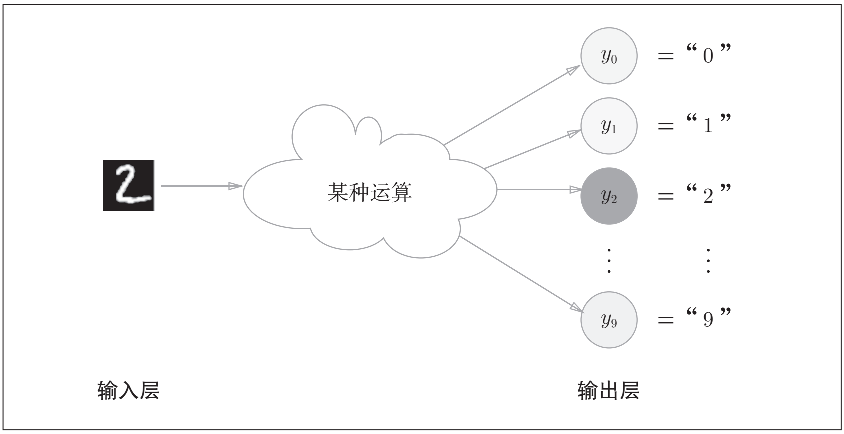


图 3-23　输出层的神经元对应各个数字

3.6　手写数字识别

介绍完神经网络的结构之后，现在我们来试着解决实际问题。这里我们来进行手写数字图像的分类。假设学习已经全部结束，我们使用学习到的参数，先实现神经网络的“推理处理”。这个推理处理也称为神经网络的**前向传播**（forward propagation）。

和求解机器学习问题的步骤（分成学习和推理两个阶段进行）一样，使用神经网络解决问题时，也需要首先使用训练数据（学习数据）进行权重参数的学习；进行推理时，使用刚才学习到的参数，对输入数据进行分类。

3.6.1 MNIST数据集

这里使用的数据集是MNIST手写数字图像集。MNIST是机器学习领域最有名的数据集之一，被应用于从简单的实验到发表的论文研究等各种场合。实际上，在阅读图像识别或机器学习的论文时，MNIST数据集经常作为实验用的数据出现。

MNIST数据集是由0到9的数字图像构成的（图3-24）。训练图像有6万张，测试图像有1万张，这些图像可以用于学习和推理。MNIST数据集的一般使用方法是，先用训练图像进行学习，再用学习到的模型度量能在多大程度上对测试图像进行正确的分类。



图3-24 MNIST图像数据集的例子

MNIST的图像数据是28像素 × 28像素的灰度图像（1通道），各个像素的取值在0到255之间。每个图像数据都相应地标有“7”“2”“1”等标签。

本书提供了便利的Python脚本`mnist.py`，该脚本支持从下载MNIST数据集到将这些数据转换成NumPy数组等处理（`mnist.py`在`dataset`目录下）。使用`mnist.py`时，当前目录必须是`ch01`、`ch02`、`ch03`、…、`ch08`目录中的一个。使用`mnist.py`中的`load_mnist()`函数，就可以按下述方式轻松读入MNIST数据。

```
import sys, os
sys.path.append(os.pardir) # 为了导入父目录中的文件而进行的设定
from dataset.mnist import load_mnist

# 第一次调用会花费几分钟……
(x_train, t_train), (x_test, t_test) = load_mnist(flatten=True,
normalize=False)

# 输出各个数据的形状
print(x_train.shape) # (60000, 784)
```

```
print(t_train.shape) # (60000,)
print(x_test.shape)  # (10000, 784)
print(t_test.shape)  # (10000,)
```

首先，为了导入父目录中的文件，进行相应的设定[1]。然后，导入dataset/mnist.py中的load_mnist函数。最后，使用load_mnist函数，读入MNIST数据集。第一次调用load_mnist函数时，因为要下载MNIST数据集，所以需要接入网络。第2次及以后的调用只需读入保存在本地的文件（pickle文件）即可，因此处理所需的时间非常短。

用来读入MNIST图像的文件在本书提供的源代码的dataset目录下。并且，我们假定了这个MNIST数据集只能从ch01、ch02、ch03、…、ch08目录中使用，因此，使用时需要从父目录（dataset目录）中导入文件，为此需要添加sys.path.append(os.pardir)语句。

load_mnist函数以“**(训练图像,训练标签)，(测试图像，测试标签)**”的形式返回读入的MNIST数据。此外，还可以像load_mnist(normalize=True, flatten=True, one_hot_label=False) 这样，设置3个参数。第1个参数normalize设置是否将输入图像正规化为0.0～1.0的值。如果将该参数设置为False，则输入图像的像素会保持原来的0～255。第2个参数flatten设置是否展开输入图像（变成一维数组）。如果将该参数设置为False，则输入图像为1 × 28 × 28的三维数组；若设置为True，则输入图像会保存为由784个元素构成的一维数组。第3个参数one_hot_label设置是否将标签保存为one-hot表示（one-hot representation）。one-hot表示是仅正确解标签为1，其余皆为0的数组，就像[0,0,1,0,0,0,0,0,0,0]这样。当one_hot_label为False时，只是像7、2这样简单保存正确解标签；当one_hot_label为True时，标签则保存为one-hot表示。

① 观察本书源代码可知，上述代码在mnist_show.py文件中。mnist_show.py文件的当前目录是ch03，但包含load_mnist()函数的mnist.py文件在dataset目录下。因此，mnist_show.py文件不能跨目录直接导入mnist.py文件。sys.path.append(os.pardir)语句实际上是把父目录deep-learning-from-scratch加入到sys.path（Python的搜索模块的路径集）中，从而可以导入deep-learning-from-scratch下的任何目录（包括dataset目录）中的任何文件。——译者注

Python有pickle这个便利的功能。这个功能可以将程序运行中的对象保存为文件。如果加载保存过的pickle文件，可以立刻复原之前程序运行中的对象。用于读入MNIST数据集的`load_mnist()`函数内部也使用了pickle功能（在第2次及以后读入时）。利用pickle功能，可以高效地完成MNIST数据的准备工作。

现在，我们试着显示MNIST图像，同时也确认一下数据。图像的显示使用PIL（Python Image Library）模块。执行下述代码后，训练图像的第一张就会显示出来，如图3-25所示（源代码在`ch03/mnist_show.py`中）。

```
import sys, os
sys.path.append(os.pardir)
import numpy as np
from dataset.mnist import load_mnist
from PIL import Image

def img_show(img):
    pil_img = Image.fromarray(np.uint8(img))
    pil_img.show()

(x_train, t_train), (x_test, t_test) = load_mnist(flatten=True,
normalize=False)
img = x_train[0]
label = t_train[0]
print(label) # 5

print(img.shape)          # (784,)
img = img.reshape(28, 28) # 把图像的形状变成原来的尺寸
print(img.shape)          # (28, 28)

img_show(img)
```

这里需要注意的是，`flatten=True`时读入的图像是以一列（一维）NumPy数组的形式保存的。因此，显示图像时，需要把它变为原来的28像素 × 28像素的形状。可以通过`reshape()`方法的参数指定期望的形状，更改NumPy数组的形状。此外，还需要把保存为NumPy数组的图像数据转换为PIL用的数据对象，这个转换处理由`Image.fromarray()`来完成。

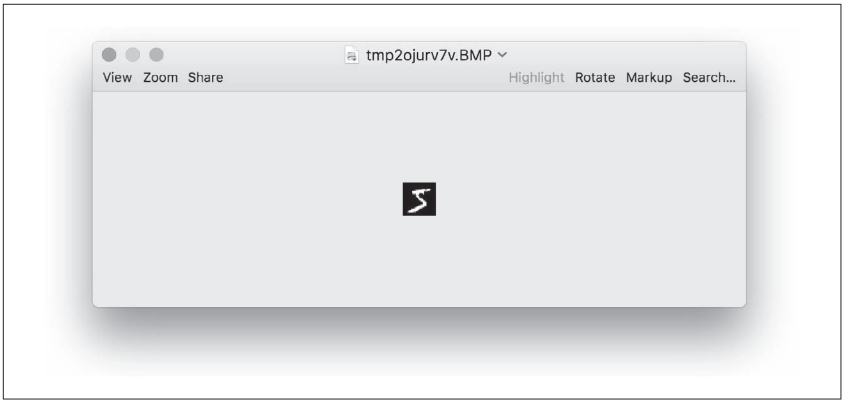


图 3-25 显示MNIST图像

3.6.2 神经网络的推理处理

下面，我们对这个MNIST数据集实现神经网络的推理处理。神经网络的输入层有784个神经元，输出层有10个神经元。输入层的784这个数字来源于图像大小的 $28 \times 28 = 784$，输出层的10这个数字来源于10类别分类（数字0到9，共10类别）。此外，这个神经网络有2个隐藏层，第1个隐藏层有50个神经元，第2个隐藏层有100个神经元。这个50和100可以设置为任何值。下面我们先定义 `get_data()`、`init_network()`、`predict()` 这3个函数（代码在 `ch03/neuralnet_mnist.py` 中）。

```
def get_data():
    (x_train, t_train), (x_test, t_test) = \
        load_mnist(normalize=True, flatten=True, one_hot_label=False)
    return x_test, t_test

def init_network():
    with open("sample_weight.pkl", 'rb') as f:
        network = pickle.load(f)

    return network

def predict(network, x):
    W1, W2, W3 = network['W1'], network['W2'], network['W3']
    b1, b2, b3 = network['b1'], network['b2'], network['b3']
```

```
    a1 = np.dot(x, W1) + b1
    z1 = sigmoid(a1)
    a2 = np.dot(z1, W2) + b2
    z2 = sigmoid(a2)
    a3 = np.dot(z2, W3) + b3
    y = softmax(a3)

    return y
```

`init_network()`会读入保存在pickle文件`sample_weight.pkl`中的学习到的权重参数[①]。这个文件中以字典变量的形式保存了权重和偏置参数。剩余的2个函数，和前面介绍的代码实现基本相同，无需再解释。现在，我们用这3个函数来实现神经网络的推理处理。然后，评价它的**识别精度**（accuracy），即能在多大程度上正确分类。

```
x, t = get_data()
network = init_network()

accuracy_cnt = 0
for i in range(len(x)):
    y = predict(network, x[i])
    p = np.argmax(y) # 获取概率最高的元素的索引
    if p == t[i]:
        accuracy_cnt += 1

print("Accuracy:" + str(float(accuracy_cnt) / len(x)))
```

首先获得MNIST数据集，生成网络。接着，用`for`语句逐一取出保存在`x`中的图像数据，用`predict()`函数进行分类。`predict()`函数以NumPy数组的形式输出各个标签对应的概率。比如输出`[0.1, 0.3, 0.2, ···, 0.04]`的数组，该数组表示“0”的概率为0.1，“1”的概率为0.3，等等。然后，我们取出这个概率列表中的最大值的索引（第几个元素的概率最高），作为预测结果。可以用`np.argmax(x)`函数取出数组中的最大值的索引，`np.argmax(x)`将获取被赋给参数`x`的数组中的最大值元素的索引。最后，比较神经网络所预测的答案和正确解标签，将回答正确的概率作为识别精度。

① 因为之前我们假设学习已经完成，所以学习到的参数被保存下来。假设保存在`sample_weight.pkl`文件中，在推理阶段，我们直接加载这些已经学习到的参数。——译者注

执行上面的代码后，会显示“Accuracy:0.9352”。这表示有93.52％的数据被正确分类了。目前我们的目标是运行学习到的神经网络，所以不讨论识别精度本身，不过以后我们会花精力在神经网络的结构和学习方法上，思考如何进一步提高这个精度。实际上，我们打算把精度提高到99％以上。

另外，在这个例子中，我们把`load_mnist`函数的参数`normalize`设置成了`True`。将`normalize`设置成`True`后，函数内部会进行转换，将图像的各个像素值除以255，使得数据的值在0.0～1.0的范围内。像这样把数据限定到某个范围内的处理称为**正规化**（normalization）。此外，对神经网络的输入数据进行某种既定的转换称为**预处理**（pre-processing）。这里，作为对输入图像的一种预处理，我们进行了正规化。

> 预处理在神经网络（深度学习）中非常实用，其有效性已在提高识别性能和学习的效率等众多实验中得到证明。在刚才的例子中，作为一种预处理，我们将各个像素值除以255，进行了简单的正规化。实际上，很多预处理都会考虑到数据的整体分布。比如，利用数据整体的均值或标准差，移动数据，使数据整体以0为中心分布，或者进行正规化，把数据的延展控制在一定范围内。除此之外，还有将数据整体的分布形状均匀化的方法，即数据**白化**（whitening）等。

3.6.3 批处理

以上就是处理MNIST数据集的神经网络的实现，现在我们来关注输入数据和权重参数的“形状”。再看一下刚才的代码实现。

下面我们使用Python解释器，输出刚才的神经网络的各层的权重的形状。

```
>>> x, _ = get_data()
>>> network = init_network()
>>> W1, W2, W3 = network['W1'], network['W2'], network['W3']
>>>
>>> x.shape
(10000, 784)
>>> x[0].shape
(784,)
>>> W1.shape
```

```
(784, 50)
>>> W2.shape
(50, 100)
>>> W3.shape
(100, 10)
```

我们通过上述结果来确认一下多维数组的对应维度的元素个数是否一致（省略了偏置）。用图表示的话，如图3-26所示。可以发现，多维数组的对应维度的元素个数确实是一致的。此外，我们还可以确认最终的结果是输出了元素个数为10 的一维数组。

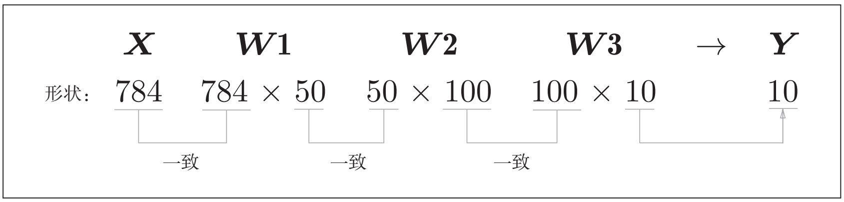


图3-26　数组形状的变化

从整体的处理流程来看，图3-26中，输入一个由784个元素（原本是一个28 × 28的二维数组）构成的一维数组后，输出一个有10个元素的一维数组。这是只输入一张图像数据时的处理流程。

现在我们来考虑打包输入多张图像的情形。比如，我们想用`predict()`函数一次性打包处理100张图像。为此，可以把$\boldsymbol{x}$的形状改为100 × 784，将100张图像打包作为输入数据。用图表示的话，如图3-27所示。

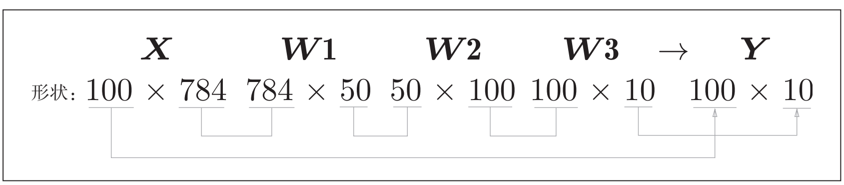


图3-27　批处理中数组形状的变化

如图3-27所示，输入数据的形状为100 × 784，输出数据的形状为100 × 10。这表示输入的100张图像的结果被一次性输出了。比如，`x[0]`和

y[0] 中保存了第0张图像及其推理结果，x[1] 和 y[1] 中保存了第1张图像及其推理结果，等等。

这种打包式的输入数据称为**批**（batch）。批有“捆”的意思，图像就如同纸币一样扎成一捆。

批处理对计算机的运算大有利处，可以大幅缩短每张图像的处理时间。那么为什么批处理可以缩短处理时间呢？这是因为大多数处理数值计算的库都进行了能够高效处理大型数组运算的最优化。并且，在神经网络的运算中，当数据传送成为瓶颈时，批处理可以减轻数据总线的负荷（严格地讲，相对于数据读入，可以将更多的时间用在计算上）。也就是说，批处理一次性计算大型数组要比分开逐步计算各个小型数组速度更快。

下面我们进行基于批处理的代码实现。这里用粗体显示与之前的实现的不同之处。

```
x, t = get_data()
network = init_network()

batch_size = 100 # 批数量
accuracy_cnt = 0

for i in range(0, len(x), batch_size):
    x_batch = x[i:i+batch_size]
    y_batch = predict(network, x_batch)
    p = np.argmax(y_batch, axis=1)
    accuracy_cnt += np.sum(p == t[i:i+batch_size])

print("Accuracy:" + str(float(accuracy_cnt) / len(x)))
```

我们来逐个解释粗体的代码部分。首先是 range() 函数。range() 函数若指定为 range(start, end)，则会生成一个由 start 到 end-1 之间的整数构成的列表。若像 range(start, end, step) 这样指定3个整数，则生成的列表中的下一个元素会增加 step 指定的值。我们来看一个例子。

```
>>> list( range(0, 10) )
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> list( range(0, 10, 3) )
[0, 3, 6, 9]
```

在range()函数生成的列表的基础上，通过x[i:i+batch_size]从输入数据中抽出批数据。x[i:i+batch_n]会取出从第i个到第i+batch_n个之间的数据。本例中是像x[0:100]、x[100:200]……这样，从头开始以100为单位将数据提取为批数据。

然后，通过argmax()获取值最大的元素的索引。不过这里需要注意的是，我们给定了参数axis=1。这指定了在100 × 10的数组中，沿着第1维方向（以第1维为轴）找到值最大的元素的索引（第0维对应第1个维度）①。这里也来看一个例子。

```
>>> x = np.array([[0.1, 0.8, 0.1], [0.3, 0.1, 0.6],
...     [0.2, 0.5, 0.3], [0.8, 0.1, 0.1]])
>>> y = np.argmax(x, axis=1)
>>> print(y)
[1 2 1 0]
```

最后，我们比较一下以批为单位进行分类的结果和实际的答案。为此，需要在NumPy数组之间使用比较运算符（==）生成由True/False构成的布尔型数组，并计算True的个数。我们通过下面的例子进行确认。

```
>>> y = np.array([1, 2, 1, 0])
>>> t = np.array([1, 2, 0, 0])
>>> print(y==t)
[True True False True]
>>> np.sum(y==t)
3
```

至此，基于批处理的代码实现就介绍完了。使用批处理，可以实现高速且高效的运算。下一章介绍神经网络的学习时，我们将把图像数据作为打包的批数据进行学习，届时也将进行和这里的批处理一样的代码实现。

① 矩阵的第0维是列方向，第1维是行方向。——译者注

3.7 小结

本章介绍了神经网络的前向传播。本章介绍的神经网络和上一章的感知机在信号的按层传递这一点上是相同的，但是，向下一个神经元发送信号时，改变信号的激活函数有很大差异。神经网络中使用的是平滑变化的sigmoid函数，而感知机中使用的是信号急剧变化的阶跃函数。这个差异对于神经网络的学习非常重要，我们将在下一章介绍。

本章所学的内容

- 神经网络中的激活函数使用平滑变化的sigmoid函数或ReLU函数。
- 通过巧妙地使用NumPy多维数组，可以高效地实现神经网络。
- 机器学习的问题大体上可以分为回归问题和分类问题。
- 关于输出层的激活函数，回归问题中一般用恒等函数，分类问题中一般用softmax函数。
- 分类问题中，输出层的神经元的数量设置为要分类的类别数。
- 输入数据的集合称为批。通过以批为单位进行推理处理，能够实现高速的运算。

第4章 神经网络的学习

本章的主题是神经网络的学习。这里所说的“学习”是指从训练数据中自动获取最优权重参数的过程。本章中，为了使神经网络能进行学习，将导入损失函数这一指标。而学习的目的就是以该损失函数为基准，找出能使它的值达到最小的权重参数。为了找出尽可能小的损失函数的值，本章我们将介绍利用了函数斜率的梯度法。

4.1 从数据中学习

神经网络的特征就是可以从数据中学习。所谓“从数据中学习”，是指可以由数据自动决定权重参数的值。这是非常了不起的事情！因为如果所有的参数都需要人工决定的话，工作量就太大了。在第2章介绍的感知机的例子中，我们对照着真值表，人工设定了参数的值，但是那时的参数只有3个。而在实际的神经网络中，参数的数量成千上万，在层数更深的深度学习中，参数的数量甚至可以上亿，想要人工决定这些参数的值是不可能的。本章将介绍神经网络的学习，即利用数据决定参数值的方法，并用Python实现对MNIST手写数字数据集的学习。

对于线性可分问题，第2章的感知机是可以利用数据自动学习的。根据“感知机收敛定理”，通过有限次数的学习，线性可分问题是可解的。但是，非线性可分问题则无法通过（自动）学习来解决。

4.1.1 数据驱动

数据是机器学习的命根子。从数据中寻找答案、从数据中发现模式、根据数据讲故事……这些机器学习所做的事情，如果没有数据的话，就无从谈起。因此，数据是机器学习的核心。这种数据驱动的方法，也可以说脱离了过往以人为中心的方法。

通常要解决某个问题，特别是需要发现某种模式时，人们一般会综合考虑各种因素后再给出回答。“这个问题好像有这样的规律性？”“不对，可能原因在别的地方。”——类似这样，人们以自己的经验和直觉为线索，通过反复试验推进工作。而机器学习的方法则极力避免人为介入，尝试从收集到的数据中发现答案（模式）。神经网络或深度学习则比以往的机器学习方法更能避免人为介入。

现在我们来思考一个具体的问题，比如如何实现数字“5”的识别。数字5是图4-1所示的手写图像，我们的目标是实现能区别是否是5的程序。这个问题看起来很简单，大家能想到什么样的算法呢？

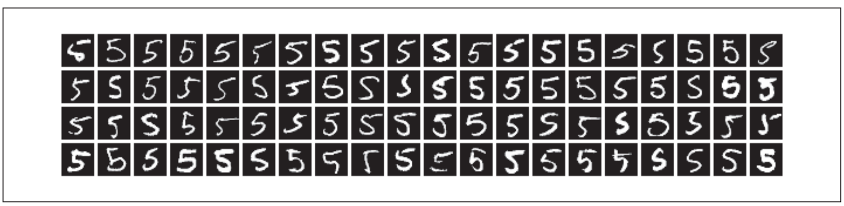


图4-1 手写数字5的例子：写法因人而异，五花八门

如果让我们自己来设计一个能将5正确分类的程序，就会意外地发现这是一个很难的问题。人可以简单地识别出5，但却很难明确说出是基于何种规律而识别出了5。此外，从图4-1中也可以看到，每个人都有不同的写字习惯，要发现其中的规律是一件非常难的工作。

因此，与其绞尽脑汁，从零开始想出一个可以识别5的算法，不如考虑通过有效利用数据来解决这个问题。一种方案是，先从图像中提取**特征量**，

再用机器学习技术学习这些特征量的模式。这里所说的“特征量”是指可以从输入数据（输入图像）中准确地提取本质数据（重要的数据）的转换器。图像的特征量通常表示为向量的形式。在计算机视觉领域，常用的特征量包括SIFT、SURF和HOG等。使用这些特征量将图像数据转换为向量，然后对转换后的向量使用机器学习中的SVM、KNN等分类器进行学习。

机器学习的方法中，由机器从收集到的数据中找出规律性。与从零开始想出算法相比，这种方法可以更高效地解决问题，也能减轻人的负担。但是需要注意的是，将图像转换为向量时使用的特征量仍是由人设计的。对于不同的问题，必须使用合适的特征量（必须设计专门的特征量），才能得到好的结果。比如，为了区分狗的脸部，人们需要考虑与用于识别5的特征量不同的其他特征量。也就是说，即使使用特征量和机器学习的方法，也需要针对不同的问题人工考虑合适的特征量。

到这里，我们介绍了两种针对机器学习任务的方法。将这两种方法用图来表示，如图4-2所示。图中还展示了神经网络（深度学习）的方法，可以看出该方法不存在人为介入。

如图4-2所示，神经网络直接学习图像本身。在第2个方法，即利用特征量和机器学习的方法中，特征量仍是由人工设计的，而在神经网络中，连图像中包含的重要特征量也都是由机器来学习的。

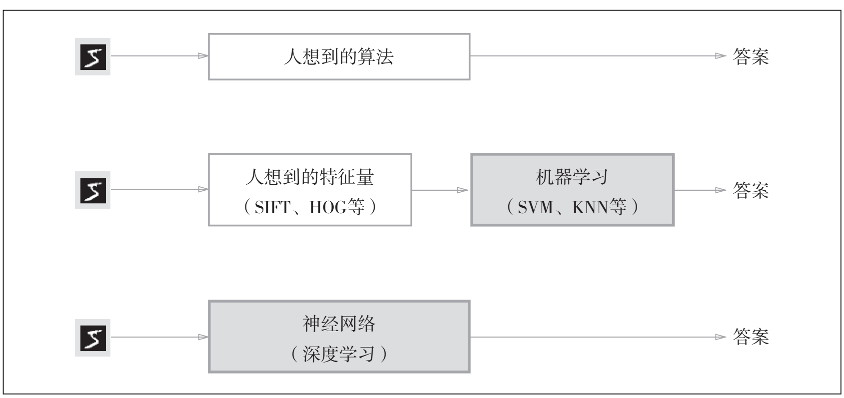


图4-2 从人工设计规则转变为由机器从数据中学习：没有人为介入的方块用灰色表示

深度学习有时也称为端到端机器学习（end-to-end machine learning）。这里所说的**端到端**是指从一端到另一端的意思，也就是从原始数据（输入）中获得目标结果（输出）的意思。

神经网络的优点是对所有的问题都可以用同样的流程来解决。比如，不管要求解的问题是识别5，还是识别狗，抑或是识别人脸，神经网络都是通过不断地学习所提供的数据，尝试发现待求解的问题的模式。也就是说，与待处理的问题无关，神经网络可以将数据直接作为原始数据，进行“端对端”的学习。

4.1.2 训练数据和测试数据

本章主要介绍神经网络的学习，不过在这之前，我们先来介绍一下机器学习中有关数据处理的一些注意事项。

机器学习中，一般将数据分为**训练数据**和**测试数据**两部分来进行学习和实验等。首先，使用训练数据进行学习，寻找最优的参数；然后，使用测试数据评价训练得到的模型的实际能力。为什么需要将数据分为训练数据和测试数据呢？因为我们追求的是模型的泛化能力。为了正确评价模型的**泛化能力**，就必须划分训练数据和测试数据。另外，训练数据也可以称为**监督数据**。

泛化能力是指处理未被观察过的数据（不包含在训练数据中的数据）的能力。获得泛化能力是机器学习的最终目标。比如，在识别手写数字的问题中，泛化能力可能会被用在自动读取明信片的邮政编码的系统上。此时，手写数字识别就必须具备较高的识别“某个人”写的字的能力。注意这里不是“特定的某个人写的特定的文字”，而是“任意一个人写的任意文字”。如果系统只能正确识别已有的训练数据，那有可能是只学习到了训练数据中的个人的习惯写法。

因此，仅仅用一个数据集去学习和评价参数，是无法进行正确评价的。这样会导致可以顺利地处理某个数据集，但无法处理其他数据集的情况。顺便说一下，只对某个数据集过度拟合的状态称为**过拟合**（over fitting）。避免过拟合也是机器学习的一个重要课题。

4.2 损失函数

如果有人问你现在有多幸福，你会如何回答呢？一般的人可能会给出诸如“还可以吧”或者“不是那么幸福”等笼统的回答。如果有人回答“我现在的幸福指数是10.23”的话，可能会把人吓一跳吧。因为他用一个数值指标来评判自己的幸福程度。

这里的幸福指数只是打个比方，实际上神经网络的学习也在做同样的事情。神经网络的学习通过某个指标表示现在的状态。然后，以这个指标为基准，寻找最优权重参数。和刚刚那位以幸福指数为指引寻找“最优人生”的人一样，神经网络以某个指标为线索寻找最优权重参数。神经网络的学习中所用的指标称为**损失函数**（loss function）。这个损失函数可以使用任意函数，但一般用均方误差和交叉熵误差等。



损失函数是表示神经网络性能的“恶劣程度”的指标，即当前的神经网络对监督数据在多大程度上不拟合，在多大程度上不一致。以“性能的恶劣程度”为指标可能会使人感到不太自然，但是如果给损失函数乘上一个负值，就可以解释为“在多大程度上不坏”，即“性能有多好”。并且，“使性能的恶劣程度达到最小”和“使性能的优良程度达到最大”是等价的，不管是用“恶劣程度”还是“优良程度”，做的事情本质上都是一样的。

4.2.1 均方误差

可以用作损失函数的函数有很多，其中最有名的是**均方误差**（mean squared error）。均方误差如下式所示。

$$E = \frac{1}{2}\sum_{k}(y_k - t_k)^2 \tag{4.1}$$

这里，y_k是表示神经网络的输出，t_k表示监督数据，k表示数据的维数。

比如，在3.6节手写数字识别的例子中，y_k、t_k是由如下10个元素构成的数据。

```
>>> y = [0.1, 0.05, 0.6, 0.0, 0.05, 0.1, 0.0, 0.1, 0.0, 0.0]
>>> t = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
```

数组元素的索引从第一个开始依次对应数字“0”“1”“2”…… 这里，神经网络的输出y是softmax函数的输出。由于softmax函数的输出可以理解为概率，因此上例表示“0”的概率是0.1，“1”的概率是0.05，“2”的概率是0.6等。t是监督数据，将正确解标签设为1，其他均设为0。这里，标签“2”为1，表示正确解是“2”。将正确解标签表示为1，其他标签表示为0的表示方法称为**one-hot表示**。

如式(4.1)所示，均方误差会计算神经网络的输出和正确解监督数据的各个元素之差的平方，再求总和。现在，我们用Python来实现这个均方误差，实现方式如下所示。

```
def mean_squared_error(y, t):
    return 0.5 * np.sum((y-t)**2)
```

这里，参数y和t是NumPy数组。代码实现完全遵照式(4.1)，因此不再具体说明。现在，我们使用这个函数，来实际地计算一下。

```
>>> # 设“2”为正确解
>>> t = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
>>>
>>> # 例1：“2”的概率最高的情况（0.6）
>>> y = [0.1, 0.05, 0.6, 0.0, 0.05, 0.1, 0.0, 0.1, 0.0, 0.0]
>>> mean_squared_error(np.array(y), np.array(t))
0.097500000000000031
>>>
>>> # 例2：“7”的概率最高的情况（0.6）
>>> y = [0.1, 0.05, 0.1, 0.0, 0.05, 0.1, 0.0, 0.6, 0.0, 0.0]
>>> mean_squared_error(np.array(y), np.array(t))
0.59750000000000003
```

这里举了两个例子。第一个例子中，正确解是“2”，神经网络的输出的最大值是“2”；第二个例子中，正确解是“2”，神经网络的输出的最大值是“7”。如实验结果所示，我们发现第一个例子的损失函数的值更小，和监督数据之间的误差较小。也就是说，均方误差显示第一个例子的输出结果与监督数据更加吻合。

4.2.2 交叉熵误差

除了均方误差之外，**交叉熵误差**（cross entropy error）也经常被用作损失函数。交叉熵误差如下式所示。

$$E = -\sum_{k} t_k \log y_k \tag{4.2}$$

这里，log表示以e为底数的自然对数（$\log_e$）。y_k是神经网络的输出，t_k是正确解标签。并且，t_k中只有正确解标签的索引为1，其他均为0（one-hot表示）。因此，式（4.2）实际上只计算对应正确解标签的输出的自然对数。比如，假设正确解标签的索引是“2”，与之对应的神经网络的输出是0.6，则交叉熵误差是$-\log 0.6 = 0.51$；若“2”对应的输出是0.1，则交叉熵误差为$-\log 0.1 = 2.30$。也就是说，交叉熵误差的值是由正确解标签所对应的输出结果决定的。

自然对数的图像如图4-3所示。

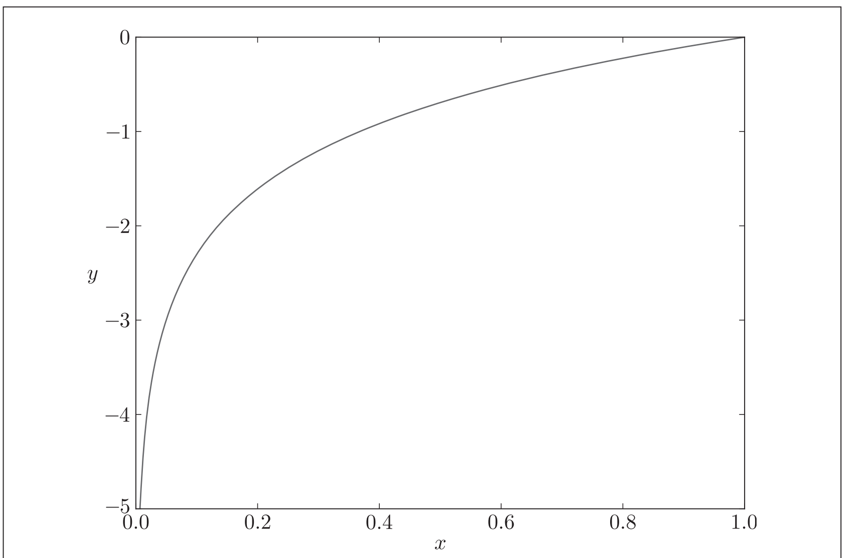


图4-3　自然对数$y = \log x$的图像

如图4-3所示，x等于1时，y为0；随着x向0靠近，y逐渐变小。因此，正确解标签对应的输出越大，式(4.2)的值越接近0；当输出为1时，交叉熵误差为0。此外，如果正确解标签对应的输出较小，则式(4.2)的值较大。

下面，我们来用代码实现交叉熵误差。

```
def cross_entropy_error(y, t):
    delta = 1e-7
    return -np.sum(t * np.log(y + delta))
```

这里，参数`y`和`t`是NumPy数组。函数内部在计算`np.log`时，加上了一个微小值`delta`。这是因为，当出现`np.log(0)`时，`np.log(0)`会变为负无限大的`-inf`，这样一来就会导致后续计算无法进行。作为保护性对策，添加一个微小值可以防止负无限大的发生。下面，我们使用`cross_entropy_error(y, t)`进行一些简单的计算。

```
>>> t = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
>>> y = [0.1, 0.05, 0.6, 0.0, 0.05, 0.1, 0.0, 0.1, 0.0, 0.0]
>>> cross_entropy_error(np.array(y), np.array(t))
0.51082545709933802
>>>
>>> y = [0.1, 0.05, 0.1, 0.0, 0.05, 0.1, 0.0, 0.6, 0.0, 0.0]
>>> cross_entropy_error(np.array(y), np.array(t))
2.3025840929945458
```

第一个例子中，正确解标签对应的输出为0.6，此时的交叉熵误差大约为0.51。第二个例子中，正确解标签对应的输出为0.1的低值，此时的交叉熵误差大约为2.3。由此可以看出，这些结果与我们前面讨论的内容是一致的。

4.2.3 mini-batch学习

机器学习使用训练数据进行学习。使用训练数据进行学习，严格来说，就是针对训练数据计算损失函数的值，找出使该值尽可能小的参数。因此，计算损失函数时必须将所有的训练数据作为对象。也就是说，如果训练数据有100个的话，我们就要把这100个损失函数的总和作为学习的指标。

前面介绍的损失函数的例子中考虑的都是针对单个数据的损失函数。如

果要求所有训练数据的损失函数的总和，以交叉熵误差为例，可以写成下面的式(4.3)。

$$E = -\frac{1}{N}\sum_{n}\sum_{k} t_{nk}\log y_{nk} \tag{4.3}$$

这里,假设数据有N个，t_{nk}表示第n个数据的第k个元素的值(y_{nk}是神经网络的输出，t_{nk}是监督数据)。式子虽然看起来有一些复杂，其实只是把求单个数据的损失函数的式(4.2)扩大到了N份数据，不过最后还要除以N进行正规化。通过除以N，可以求单个数据的“平均损失函数”。通过这样的平均化，可以获得和训练数据的数量无关的统一指标。比如，即便训练数据有1000个或10000个，也可以求得单个数据的平均损失函数。

另外，MNIST数据集的训练数据有60000个，如果以全部数据为对象求损失函数的和，则计算过程需要花费较长的时间。再者，如果遇到大数据，数据量会有几百万、几千万之多，这种情况下以全部数据为对象计算损失函数是不现实的。因此，我们从全部数据中选出一部分，作为全部数据的“近似”。神经网络的学习也是从训练数据中选出一批数据(称为mini-batch,小批量)，然后对每个mini-batch进行学习。比如，从60000个训练数据中随机选择100笔，再用这100笔数据进行学习。这种学习方式称为**mini-batch学习**。

下面我们来编写从训练数据中随机选择指定个数的数据的代码，以进行mini-batch学习。在这之前，先来看一下用于读入MNIST数据集的代码。

```
import sys, os
sys.path.append(os.pardir)
import numpy as np
from dataset.mnist import load_mnist

(x_train, t_train), (x_test, t_test) = \
    load_mnist(normalize=True, one_hot_label=True)

print(x_train.shape) # (60000, 784)
print(t_train.shape) # (60000, 10)
```

第3章介绍过，`load_mnist`函数是用于读入MNIST数据集的函数。这个函数在本书提供的脚本`dataset/mnist.py`中，它会读入训练数据和测试数据。

读入数据时，通过设定参数one_hot_label=True，可以得到one-hot表示（即仅正确解标签为1，其余为0的数据结构）。

读入上面的MNIST数据后，训练数据有60000个，输入数据是784维（28×28）的图像数据，监督数据是10维的数据。因此，上面的x_train、t_train的形状分别是(60000, 784)和(60000, 10)。

那么，如何从这个训练数据中随机抽取10笔数据呢？我们可以使用NumPy的np.random.choice()，写成如下形式。

```
train_size = x_train.shape[0]
batch_size = 10
batch_mask = np.random.choice(train_size, batch_size)
x_batch = x_train[batch_mask]
t_batch = t_train[batch_mask]
```

使用np.random.choice()可以从指定的数字中随机选择想要的数字。比如，np.random.choice(60000, 10)会从0到59999之间随机选择10个数字。如下面的实际代码所示，我们可以得到一个包含被选数据的索引的数组。

```
>>> np.random.choice(60000, 10)
array([ 8013, 14666, 58210, 23832, 52091, 10153, 8107, 19410, 27260,
21411])
```

之后，我们只需指定这些随机选出的索引，取出mini-batch，然后使用这个mini-batch计算损失函数即可。

计算电视收视率时，并不会统计所有家庭的电视机，而是仅以那些被选中的家庭为统计对象。比如，通过从关东地区随机选择1000个家庭计算收视率，可以近似地求得关东地区整体的收视率。这1000个家庭的收视率，虽然严格上不等于整体的收视率，但可以作为整体的一个近似值。和收视率一样，mini-batch的损失函数也是利用一部分样本数据来近似地计算整体。也就是说，用随机选择的小批量数据（mini-batch）作为全体训练数据的近似值。

4.2.4 mini-batch版交叉熵误差的实现

如何实现对应mini-batch的交叉熵误差呢？只要改良一下之前实现的对应单个数据的交叉熵误差就可以了。这里，我们来实现一个可以同时处理单个数据和批量数据（数据作为batch集中输入）两种情况的函数。

```
def cross_entropy_error(y, t):
    if y.ndim == 1:
        t = t.reshape(1, t.size)
        y = y.reshape(1, y.size)

    batch_size = y.shape[0]
    return -np.sum(t * np.log(y + 1e-7)) / batch_size
```

这里，y是神经网络的输出，t是监督数据。y的维度为1时，即求单个数据的交叉熵误差时，需要改变数据的形状。并且，当输入为mini-batch时，要用batch的个数进行正规化，计算单个数据的平均交叉熵误差。

此外，当监督数据是标签形式（非one-hot表示，而是像“2”“7”这样的标签）时，交叉熵误差可通过如下代码实现。

```
def cross_entropy_error(y, t):
    if y.ndim == 1:
        t = t.reshape(1, t.size)
        y = y.reshape(1, y.size)

    batch_size = y.shape[0]
    return -np.sum(np.log(y[np.arange(batch_size), t] + 1e-7)) / batch_size
```

实现的要点是，由于one-hot表示中t为0的元素的交叉熵误差也为0，因此针对这些元素的计算可以忽略。换言之，如果可以获得神经网络在正确解标签处的输出，就可以计算交叉熵误差。因此，t为one-hot表示时通过`t * np.log(y)`计算的地方，在t为标签形式时，可用`np.log( y[np.arange(batch_size), t] )`实现相同的处理（为了便于观察，这里省略了微小值1e-7）。

作为参考，简单介绍一下`np.log( y[np.arange(batch_size), t] )`。`np.arange(batch_size)`会生成一个从0到`batch_size-1`的数组。比如当`batch_size`为5时，`np.arange(batch_size)`会生成一个NumPy 数组`[0, 1, 2, 3, 4]`。因为

t中标签是以[2, 7, 0, 9, 4]的形式存储的，所以y[np.arange(batch_size), t]能抽出各个数据的正确解标签对应的神经网络的输出（在这个例子中，y[np.arange(batch_size), t]会生成NumPy数组[y[0,2], y[1,7], y[2,0], y[3,9], y[4,4]]）。

4.2.5 为何要设定损失函数

上面我们讨论了损失函数，可能有人要问：“为什么要导入损失函数呢？”以数字识别任务为例，我们想获得的是能提高识别精度的参数，特意再导入一个损失函数不是有些重复劳动吗？也就是说，既然我们的目标是获得使识别精度尽可能高的神经网络，那不是应该把识别精度作为指标吗？

对于这一疑问，我们可以根据“导数”在神经网络学习中的作用来回答。下一节中会详细说到，在神经网络的学习中，寻找最优参数（权重和偏置）时，要寻找使损失函数的值尽可能小的参数。为了找到使损失函数的值尽可能小的地方，需要计算参数的导数（确切地讲是梯度），然后以这个导数为指引，逐步更新参数的值。

假设有一个神经网络，现在我们来关注这个神经网络中的某一个权重参数。此时，对该权重参数的损失函数求导，表示的是“如果稍微改变这个权重参数的值，损失函数的值会如何变化”。如果导数的值为负，通过使该权重参数向正方向改变，可以减小损失函数的值；反过来，如果导数的值为正，则通过使该权重参数向负方向改变，可以减小损失函数的值。不过，当导数的值为0时，无论权重参数向哪个方向变化，损失函数的值都不会改变，此时该权重参数的更新会停在此处。

之所以不能用识别精度作为指标，是因为这样一来绝大多数地方的导数都会变为0，导致参数无法更新。话说得有点多了，我们来总结一下上面的内容。

> 在进行神经网络的学习时，不能将识别精度作为指标。因为如果以识别精度为指标，则参数的导数在绝大多数地方都会变为0。

为什么用识别精度作为指标时，参数的导数在绝大多数地方都会变成0

呢？为了回答这个问题，我们来思考另一个具体例子。假设某个神经网络正确识别出了100笔训练数据中的32笔，此时识别精度为32 %。如果以识别精度为指标，即使稍微改变权重参数的值，识别精度也仍将保持在32 %，不会出现变化。也就是说，仅仅微调参数，是无法改善识别精度的。即便识别精度有所改善，它的值也不会像32.0123 · · · %这样连续变化，而是变为33 %、34 %这样的不连续的、离散的值。而如果把损失函数作为指标，则当前损失函数的值可以表示为0.92543 · · · 这样的值。并且，如果稍微改变一下参数的值，对应的损失函数也会像0.93432 · · · 这样发生连续性的变化。

识别精度对微小的参数变化基本上没有什么反应，即便有反应，它的值也是不连续地、突然地变化。作为激活函数的阶跃函数也有同样的情况。出于相同的原因，如果使用阶跃函数作为激活函数，神经网络的学习将无法进行。如图4-4所示，阶跃函数的导数在绝大多数地方（除了0以外的地方）均为0。也就是说，如果使用了阶跃函数，那么即便将损失函数作为指标，参数的微小变化也会被阶跃函数抹杀，导致损失函数的值不会产生任何变化。

阶跃函数就像“竹筒敲石”一样，只在某个瞬间产生变化。而sigmoid函数，如图4-4所示，不仅函数的输出（竖轴的值）是连续变化的，曲线的斜率（导数）也是连续变化的。也就是说，sigmoid函数的导数在任何地方都不为0。这对神经网络的学习非常重要。得益于这个斜率不会为0的性质，神经网络的学习得以正确进行。

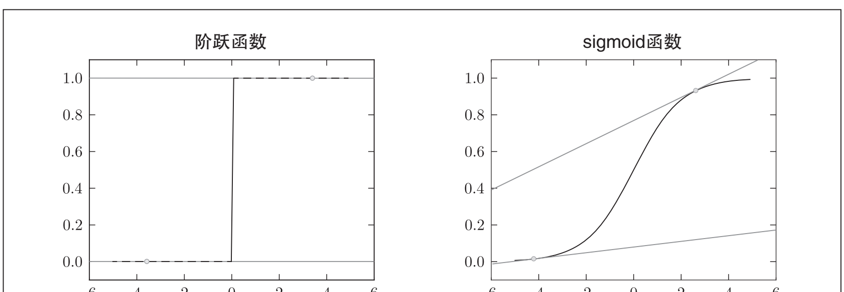


图4-4 阶跃函数和sigmoid函数：阶跃函数的斜率在绝大多数地方都为**0**，而sigmoid函数的斜率（切线）不会为**0**

4.3 数值微分

梯度法使用梯度的信息决定前进的方向。本节将介绍梯度是什么、有什么性质等内容。在这之前，我们先来介绍一下导数。

4.3.1 导数

假如你是全程马拉松选手，在开始的10分钟内跑了2千米。如果要计算此时的奔跑速度，则为$2/10 = 0.2$[千米 / 分]。也就是说，你以1分钟前进0.2千米的速度（变化）奔跑。

在这个马拉松的例子中，我们计算了“奔跑的距离”相对于“时间”发生了多大变化。不过，这个10分钟跑2千米的计算方式，严格地讲，计算的是10分钟内的平均速度。而导数表示的是某个瞬间的变化量。因此，将10分钟这一时间段尽可能地缩短，比如计算前1分钟奔跑的距离、前1秒钟奔跑的距离、前0.1秒钟奔跑的距离……这样就可以获得某个瞬间的变化量（某个瞬时速度）。

综上，导数就是表示某个瞬间的变化量。它可以定义成下面的式子。

$$\frac{\mathrm{d}f(x)}{\mathrm{d}x} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{4.4}$$

式（4.4）表示的是函数的导数。左边的符号$\frac{\mathrm{d}f(x)}{\mathrm{d}x}$表示$f(x)$关于$x$的导数，即$f(x)$相对于$x$的变化程度。式（4.4）表示的导数的含义是，$x$的“微小变化”将导致函数$f(x)$的值在多大程度上发生变化。其中，表示微小变化的$h$无限趋近0，表示为$\lim\limits_{h \to 0}$。

接下来，我们参考式（4.4），来实现求函数的导数的程序。如果直接实现式（4.4）的话，向h中赋入一个微小值，就可以计算出来了。比如，下面的实现如何？

```
# 不好的实现示例
def numerical_diff(f, x):
```

```
    h = 10e-50
    return (f(x+h) - f(x)) / h
```

函数`numerical_diff(f, x)`的名称来源于**数值微分**[①]的英文numerical differentiation。这个函数有两个参数，即“函数`f`”和“传给函数`f`的参数`x`”。乍一看这个实现没有问题，但是实际上这段代码有两处需要改进的地方。

在上面的实现中，因为想把尽可能小的值赋给`h`（可以话，想让`h`无限接近0），所以`h`使用了`10e-50`（有50个连续的0的“0.00⋯1”）这个微小值。但是，这样反而产生了**舍入误差**（rounding error）。所谓舍入误差，是指因省略小数的精细部分的数值（比如，小数点第8位以后的数值）而造成最终的计算结果上的误差。比如，在Python中，舍入误差可如下表示。

```
>>> np.float32(1e-50)
0.0
```

如上所示，如果用`float32`类型（32位的浮点数）来表示`1e-50`，就会变成0.0，无法正确表示出来。也就是说，使用过小的值会造成计算机出现计算上的问题。这是第一个需要改进的地方，即将微小值`h`改为10^{-4}。使用10^{-4}就可以得到正确的结果。

第二个需要改进的地方与函数`f`的差分有关。虽然上述实现中计算了函数`f`在`x+h`和`x`之间的差分，但是必须注意到，这个计算从一开始就有误差。如图4-5所示，“真的导数”对应函数在x处的斜率（称为切线），但上述实现中计算的导数对应的是$(x+h)$和x之间的斜率。因此，真的导数（真的切线）和上述实现中得到的导数的值在严格意义上并不一致。这个差异的出现是因为h不可能无限接近0。

如图4-5所示，数值微分含有误差。为了减小这个误差，我们可以计算函数f在$(x+h)$和$(x-h)$之间的差分。因为这种计算方法以x为中心，计算它左右两边的差分，所以也称为**中心差分**（而$(x+h)$和x之间的差分称为**前向差分**）。下面，我们基于上述两个要改进的点来实现数值微分（数值梯度）。

① 所谓数值微分就是用数值方法近似求解函数的导数的过程。——译者注

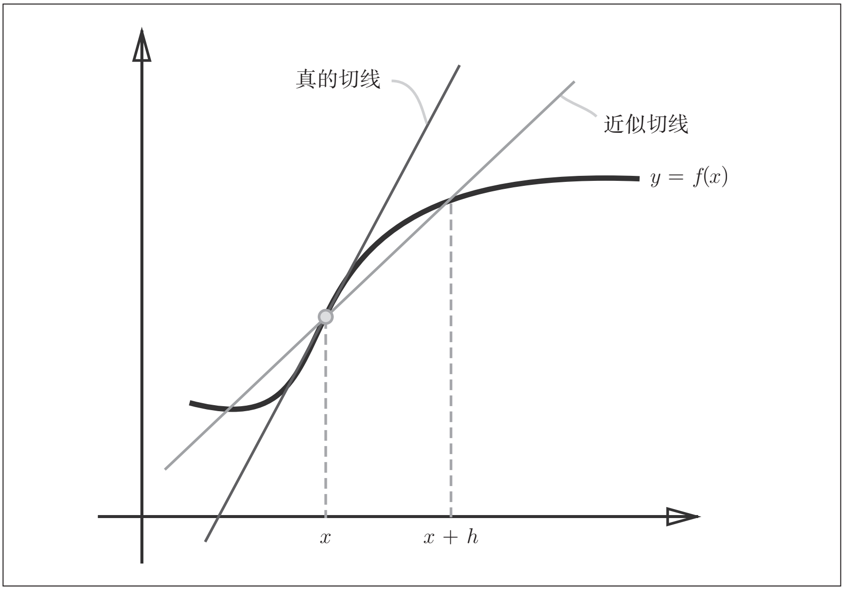


图4-5 真的导数（真的切线）和数值微分（近似切线）的值不同

```
def numerical_diff(f, x):
    h = 1e-4 # 0.0001
    return (f(x+h) - f(x-h)) / (2*h)
```

如上所示，利用微小的差分求导数的过程称为**数值微分**（numerical differentiation）。而基于数学式的推导求导数的过程，则用“**解析性**”（analytic）一词，称为“解析性求解”或者“解析性求导”。比如，$y = x^2$的导数，可以通过$\frac{\mathrm{d}y}{\mathrm{d}x} = 2x$解析性地求解出来。因此，当$x = 2$时，$y$的导数为4。解析性求导得到的导数是不含误差的“真的导数”。

4.3.2 数值微分的例子

现在我们试着用上述的数值微分对简单函数进行求导。先来看一个由下式表示的2次函数。

$$y = 0.01x^2 + 0.1x \tag{4.5}$$

用Python来实现式(4.5)，如下所示。

```
def function_1(x):
    return 0.01*x**2 + 0.1*x
```

接下来，我们来绘制这个函数的图像。画图所用的代码如下，生成的图像如图4-6所示。

```
import numpy as np
import matplotlib.pylab as plt

x = np.arange(0.0, 20.0, 0.1) # 以0.1为单位，从0到20的数组x
y = function_1(x)
plt.xlabel("x")
plt.ylabel("f(x)")
plt.plot(x, y)
plt.show()
```

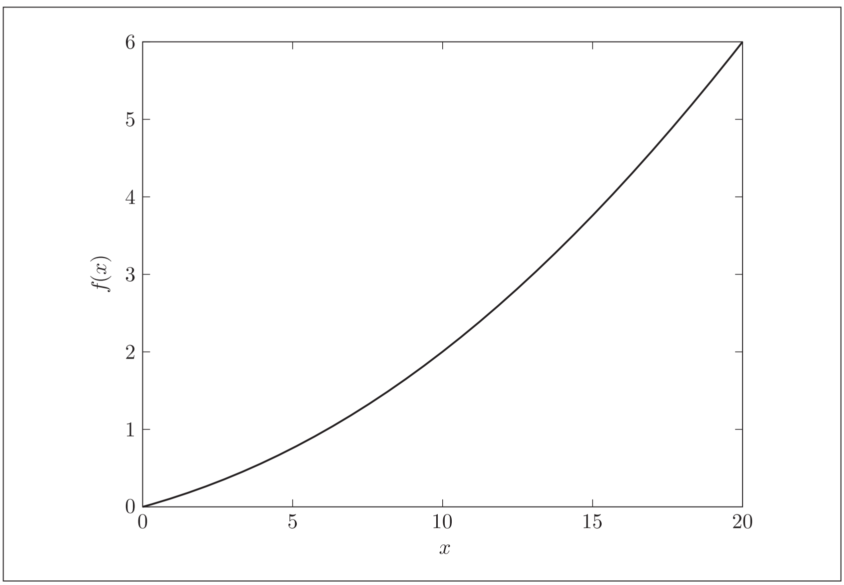


图4-6　$f(x) = 0.01x^2 + 0.1x$的图像

我们来计算一下这个函数在$x = 5$和$x = 10$处的导数。

```
>>> numerical_diff(function_1, 5)
0.1999999999990898
>>> numerical_diff(function_1, 10)
0.2999999999986347
```

这里计算的导数是$f(x)$相对于x的变化量，对应函数的斜率。另外，$f(x)=0.01x^2+0.1x$的解析解是$\frac{\mathrm{d}f(x)}{\mathrm{d}x}=0.02x+0.1$。因此，在$x=5$和$x=10$处，“真的导数”分别为0.2和0.3。和上面的结果相比，我们发现虽然严格意义上它们并不一致，但误差非常小。实际上，误差小到基本上可以认为它们是相等的。

现在，我们用上面的数值微分的值作为斜率，画一条直线。结果如图4-7所示，可以确认这些直线确实对应函数的切线（源代码在ch04/gradient_1d.py中）。

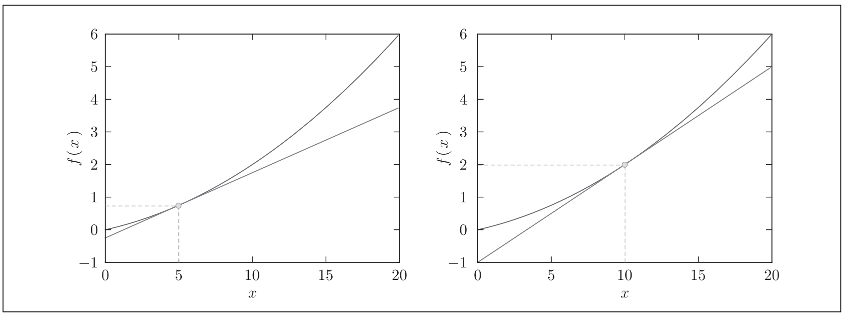


图4-7　$x=5$、$x=10$处的切线：直线的斜率使用数值微分的值

4.3.3　偏导数

接下来，我们看一下式(4.6)表示的函数。虽然它只是一个计算参数的平方和的简单函数，但是请注意和上例不同的是，这里有两个变量。

$$f(x_0,x_1)=x_0^2+x_1^2 \tag{4.6}$$

这个式子可以用Python来实现，如下所示。

```
def function_2(x):
```

```
    return x[0]**2 + x[1]**2
    # 或者return np.sum(x**2)
```

这里，我们假定向参数输入了一个NumPy数组。函数的内部实现比较简单，先计算NumPy数组中各个元素的平方，再求它们的和（np.sum(x**2)也可以实现同样的处理）。我们来画一下这个函数的图像。结果如图4-8所示，是一个三维图像。

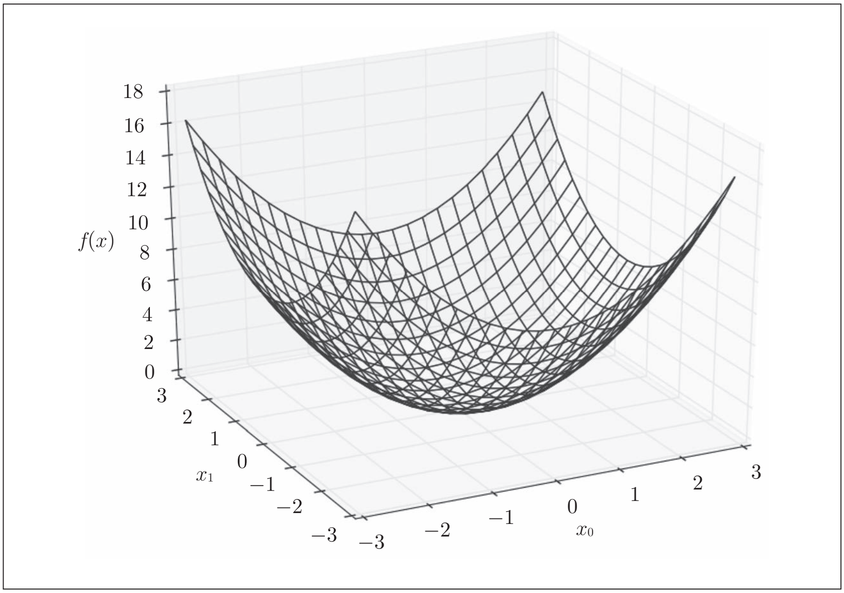


图4-8 $f(x_0, x_1) = x_0^2 + x_1^2$ 的图像

现在我们来求式(4.6)的导数。这里需要注意的是，式(4.6)有两个变量，所以有必要区分对哪个变量求导数，即对x_0和x_1两个变量中的哪一个求导数。另外，我们把这里讨论的有多个变量的函数的导数称为**偏导数**。用数学式表示的话，可以写成$\frac{\partial f}{\partial x_0}$、$\frac{\partial f}{\partial x_1}$。

怎么求偏导数呢？我们先试着解一下下面两个关于偏导数的问题。

问题1：求 $x_0=3, x_1=4$ 时，关于 x_0 的偏导数 $\frac{\partial f}{\partial x_0}$ 。

```
>>> def function_tmp1(x0):
...     return x0*x0 + 4.0**2.0
...
>>> numerical_diff(function_tmp1, 3.0)
6.00000000000378
```

问题2：求 $x_0=3, x_1=4$ 时，关于 x_1 的偏导数 $\frac{\partial f}{\partial x_1}$ 。

```
>>> def function_tmp2(x1):
...     return 3.0**2.0 + x1*x1
...
>>> numerical_diff(function_tmp2, 4.0)
7.999999999999119
```

在这些问题中，我们定义了一个只有一个变量的函数，并对这个函数进行了求导。例如，问题1中，我们定义了一个固定 $x_1=4$ 的新函数，然后对只有变量 x_0 的函数应用了求数值微分的函数。从上面的计算结果可知，问题1的答案是6.00000000000378，问题2的答案是7.999999999999119，和解析解的导数基本一致。

像这样，偏导数和单变量的导数一样，都是求某个地方的斜率。不过，偏导数需要将多个变量中的某一个变量定为目标变量，并将其他变量固定为某个值。在上例的代码中，为了将目标变量以外的变量固定到某些特定的值上，我们定义了新函数。然后，对新定义的函数应用了之前的求数值微分的函数，得到偏导数。

4.4　梯度

在刚才的例子中，我们按变量分别计算了 x_0 和 x_1 的偏导数。现在，我们希望一起计算 x_0 和 x_1 的偏导数。比如，我们来考虑求 $x_0=3, x_1=4$ 时 (x_0, x_1) 的偏导数 $\left(\frac{\partial f}{\partial x_0}, \frac{\partial f}{\partial x_1}\right)$。另外，像 $\left(\frac{\partial f}{\partial x_0}, \frac{\partial f}{\partial x_1}\right)$ 这样的由全部变量的偏导数汇总而成的向量称为**梯度**（gradient）。梯度可以像下面这样来实现。

```
def numerical_gradient(f, x):
    h = 1e-4 # 0.0001
    grad = np.zeros_like(x) # 生成和x形状相同的数组

    for idx in range(x.size):
        tmp_val = x[idx]
        # f(x+h)的计算
        x[idx] = tmp_val + h
        fxh1 = f(x)

        # f(x-h)的计算
        x[idx] = tmp_val - h
        fxh2 = f(x)

        grad[idx] = (fxh1 - fxh2) / (2*h)
        x[idx] = tmp_val # 还原值

    return grad
```

函数 numerical_gradient(f, x) 的实现看上去有些复杂，但它执行的处理和求单变量的数值微分基本没有区别。需要补充说明一下的是，np.zeros_like(x) 会生成一个形状和 x 相同、所有元素都为 0 的数组。

函数 numerical_gradient(f, x) 中，参数 f 为函数，x 为 NumPy 数组，该函数对 NumPy 数组 x 的各个元素求数值微分。现在，我们用这个函数实际计算一下梯度。这里我们求点 $(3,4)$、$(0,2)$、$(3,0)$ 处的梯度。

```
>>> numerical_gradient(function_2, np.array([3.0, 4.0]))
array([ 6.,  8.])①
>>> numerical_gradient(function_2, np.array([0.0, 2.0]))
array([ 0.,  4.])
>>> numerical_gradient(function_2, np.array([3.0, 0.0]))
array([ 6.,  0.])
```

像这样，我们可以计算 (x_0, x_1) 在各点处的梯度。上例中，点 $(3,4)$ 处的梯度是 $(6,8)$、点 $(0,2)$ 处的梯度是 $(0,4)$、点 $(3,0)$ 处的梯度是 $(6,0)$。这个梯度意味着什么呢？为了更好地理解，我们把 $f(x_0 + x_1) = x_0^2 + x_1^2$ 的梯度画在图上。不过，这里我们画的是元素值为负梯度[2] 的向量（源代码在 ch04/gradient_2d.py 中）。

① 实际上，虽然求到的值是 [6.0000000000037801, 7.9999999999991189]，但实际输出的是 [6., 8.]。这是因为在输出 NumPy 数组时，数值会被改成"易读"的形式。

② 后面我们将会看到，负梯度方向是梯度法中变量的更新方向。——译者注

如图4-9所示，$f(x_0+x_1)=x_0^2+x_1^2$ 的梯度呈现为有向向量(箭头)。观察图4-9，我们发现梯度指向函数 $f(x_0,x_1)$ 的“最低处”(最小值)，就像指南针一样，所有的箭头都指向同一点。其次，我们发现离“最低处”越远，箭头越大。

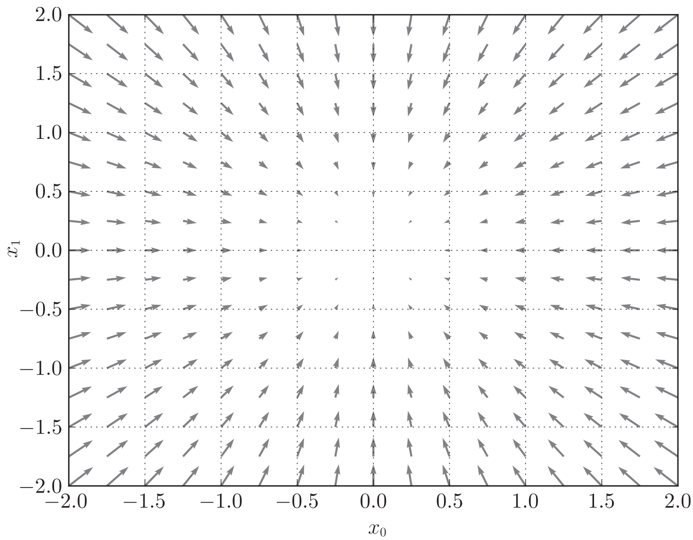


图4-9 $f(x_0,x_1)=x_0^2+x_1^2$ 的梯度

虽然图4-9中的梯度指向了最低处，但并非任何时候都这样。实际上，梯度会指向各点处的函数值降低的方向。更严格地讲，梯度指示的方向是各点处的函数值减小最多的方向[①]。这是一个非常重要的性质，请一定牢记！

4.4.1 梯度法

机器学习的主要任务是在学习时寻找最优参数。同样地，神经网络也必须在学习时找到最优参数(权重和偏置)。这里所说的最优参数是指损失函数

① 高等数学告诉我们，方向导数 = cos(θ) × 梯度(θ是方向导数的方向与梯度方向的夹角)。因此，所有的下降方向中，梯度方向下降最多。——译者注

取最小值时的参数。但是，一般而言，损失函数很复杂，参数空间庞大，我们不知道它在何处能取得最小值。而通过巧妙地使用梯度来寻找函数最小值（或者尽可能小的值）的方法就是梯度法。

这里需要注意的是，梯度表示的是各点处的函数值减小最多的方向。因此，无法保证梯度所指的方向就是函数的最小值或者真正应该前进的方向。实际上，在复杂的函数中，梯度指示的方向基本上都不是函数值最小处。

函数的极小值、最小值以及被称为**鞍点**（saddle point）的地方，梯度为0。极小值是局部最小值，也就是限定在某个范围内的最小值。鞍点是从某个方向上看是极大值，从另一个方向上看则是极小值的点。虽然梯度法是要寻找梯度为0的地方，但是那个地方不一定就是最小值（也有可能是极小值或者鞍点）。此外，当函数很复杂且呈扁平状时，学习可能会进入一个（几乎）平坦的地区，陷入被称为"学习高原"的无法前进的停滞期。

虽然梯度的方向并不一定指向最小值，但沿着它的方向能够最大限度地减小函数的值。因此，在寻找函数的最小值（或者尽可能小的值）的位置的任务中，要以梯度的信息为线索，决定前进的方向。

此时梯度法就派上用场了。在梯度法中，函数的取值从当前位置沿着梯度方向前进一定距离，然后在新的地方重新求梯度，再沿着新梯度方向前进，如此反复，不断地沿梯度方向前进。像这样，通过不断地沿梯度方向前进，逐渐减小函数值的过程就是**梯度法**（gradient method）。梯度法是解决机器学习中最优化问题的常用方法，特别是在神经网络的学习中经常被使用。

根据目的是寻找最小值还是最大值，梯度法的叫法有所不同。严格地讲，寻找最小值的梯度法称为**梯度下降法**（gradient descent method），寻找最大值的梯度法称为**梯度上升法**（gradient ascent method）。但是通过反转损失函数的符号，求最小值的问题和求最大值的问题会变成相同的问题，因此"下降"还是"上升"的差异本质上并不重要。一般来说，神经网络（深度学习）中，梯度法主要是指梯度下降法。

现在，我们尝试用数学式来表示梯度法，如式(4.7)所示。

$$\begin{aligned} x_0 &= x_0 - \eta \frac{\partial f}{\partial x_0} \\ x_1 &= x_1 - \eta \frac{\partial f}{\partial x_1} \end{aligned} \tag{4.7}$$

式(4.7)的η表示更新量，在神经网络的学习中，称为**学习率**(learning rate)。学习率决定在一次学习中，应该学习多少，以及在多大程度上更新参数。

式(4.7)是表示更新一次的式子，这个步骤会反复执行。也就是说，每一步都按式(4.7)更新变量的值，通过反复执行此步骤，逐渐减小函数值。虽然这里只展示了有两个变量时的更新过程，但是即便增加变量的数量，也可以通过类似的式子(各个变量的偏导数)进行更新。

学习率需要事先确定为某个值，比如0.01或0.001。一般而言，这个值过大或过小，都无法抵达一个"好的位置"。在神经网络的学习中，一般会一边改变学习率的值，一边确认学习是否正确进行了。

下面，我们用Python来实现梯度下降法。如下所示，这个实现很简单。

```
def gradient_descent(f, init_x, lr=0.01, step_num=100):
    x = init_x

    for i in range(step_num):
        grad = numerical_gradient(f, x)
        x -= lr * grad

    return x
```

参数`f`是要进行最优化的函数，`init_x`是初始值，`lr`是学习率learning rate，`step_num`是梯度法的重复次数。`numerical_gradient(f,x)`会求函数的梯度，用该梯度乘以学习率得到的值进行更新操作，由`step_num`指定重复的次数。

使用这个函数可以求函数的极小值，顺利的话，还可以求函数的最小值。下面，我们就来尝试解决下面这个问题。

问题：请用梯度法求 $f(x_0+x_1)=x_0^2+x_1^2$ 的最小值。

```
>>> def function_2(x):
...     return x[0]**2 + x[1]**2
...
>>> init_x = np.array([-3.0, 4.0])
>>> gradient_descent(function_2, init_x=init_x, lr=0.1, step_num=100)
array([ -6.11110793e-10,   8.14814391e-10])
```

这里，设初始值为`(-3.0, 4.0)`，开始使用梯度法寻找最小值。最终的结果是`(-6.1e-10, 8.1e-10)`，非常接近`(0, 0)`。实际上，真的最小值就是`(0, 0)`，所以说通过梯度法我们基本得到了正确结果。如果用图来表示梯度法的更新过程，则如图4-10所示。可以发现，原点处是最低的地方，函数的取值一点点在向其靠近。这个图的源代码在`ch04/gradient_method.py`中（但`ch04/gradient_method.py`不显示表示等高线的虚线）。

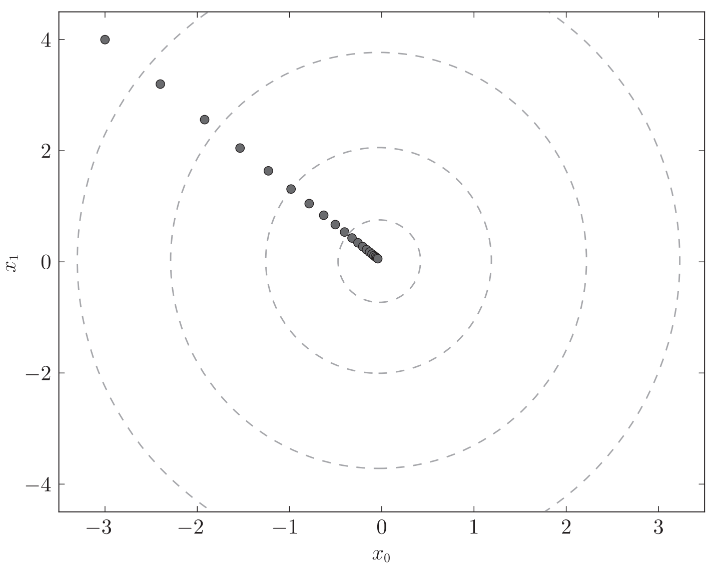


图4-10　$f(x_0,x_1)=x_0^2+x_1^2$ 的梯度法的更新过程：虚线是函数的等高线

前面说过，学习率过大或者过小都无法得到好的结果。我们来做个实验验证一下。

```
# 学习率过大的例子：lr=10.0
>>> init_x = np.array([-3.0, 4.0])
>>> gradient_descent(function_2, init_x=init_x, lr=10.0, step_num=100)
array([ -2.58983747e+13,  -1.29524862e+12])

# 学习率过小的例子：lr=1e-10
>>> init_x = np.array([-3.0, 4.0])
>>> gradient_descent(function_2, init_x=init_x, lr=1e-10, step_num=100)
array([-2.99999994,  3.99999992])
```

实验结果表明，学习率过大的话，会发散成一个很大的值；反过来，学习率过小的话，基本上没怎么更新就结束了。也就是说，设定合适的学习率是一个很重要的问题。

像学习率这样的参数称为**超参数**。这是一种和神经网络的参数（权重和偏置）性质不同的参数。相对于神经网络的权重参数是通过训练数据和学习算法自动获得的，学习率这样的超参数则是人工设定的。一般来说，超参数需要尝试多个值，以便找到一种可以使学习顺利进行的设定。

4.4.2　神经网络的梯度

神经网络的学习也要求梯度。这里所说的梯度是指损失函数关于权重参数的梯度。比如，有一个只有一个形状为2×3的权重$\boldsymbol{W}$的神经网络，损失函数用L表示。此时，梯度可以用$\frac{\partial L}{\partial \boldsymbol{W}}$表示。用数学式表示的话，如下所示。

$$
\begin{aligned}
\boldsymbol{W} &= \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{pmatrix} \\
\frac{\partial L}{\partial \boldsymbol{W}} &= \begin{pmatrix} \frac{\partial L}{\partial w_{11}} & \frac{\partial L}{\partial w_{12}} & \frac{\partial L}{\partial w_{13}} \\ \frac{\partial L}{\partial w_{21}} & \frac{\partial L}{\partial w_{22}} & \frac{\partial L}{\partial w_{23}} \end{pmatrix}
\end{aligned}
\tag{4.8}
$$

$\frac{\partial L}{\partial \boldsymbol{W}}$的元素由各个元素关于$\boldsymbol{W}$的偏导数构成。比如，第1行第1列的元

素 $\frac{\partial L}{\partial w_{11}}$ 表示当 w_{11} 稍微变化时，损失函数 L 会发生多大变化。这里的重点是，$\frac{\partial L}{\partial \boldsymbol{W}}$ 的形状和 $\boldsymbol{W}$ 相同。实际上，式(4.8)中的 $\boldsymbol{W}$ 和 $\frac{\partial L}{\partial \boldsymbol{W}}$ 都是 2×3 的形状。

下面，我们以一个简单的神经网络为例，来实现求梯度的代码。为此，我们要实现一个名为 simpleNet 的类（源代码在 ch04/gradient_simplenet.py 中）。

```
import sys, os
sys.path.append(os.pardir)
import numpy as np
from common.functions import softmax, cross_entropy_error
from common.gradient import numerical_gradient

class simpleNet:
    def __init__(self):
        self.W = np.random.randn(2,3) # 用高斯分布进行初始化

    def predict(self, x):
        return np.dot(x, self.W)

    def loss(self, x, t):
        z = self.predict(x)
        y = softmax(z)
        loss = cross_entropy_error(y, t)

        return loss
```

这里使用了 common/functions.py 中的 softmax 和 cross_entropy_error 方法，以及 common/gradient.py 中的 numerical_gradient 方法。simpleNet 类只有一个实例变量，即形状为 2×3 的权重参数。它有两个方法，一个是用于预测的 predict(x)，另一个是用于求损失函数值的 loss(x,t)。这里参数 x 接收输入数据，t 接收正确解标签。现在我们来试着用一下这个 simpleNet。

```
>>> net = simpleNet()
>>> print(net.W) # 权重参数
[[ 0.47355232   0.9977393    0.84668094],
 [ 0.85557411   0.03563661   0.69422093]])
>>>
>>> x = np.array([0.6, 0.9])
>>> p = net.predict(x)
>>> print(p)
```

```
[ 1.05414809  0.63071653  1.1328074]
>>> np.argmax(p) # 最大值的索引
2
>>>
>>> t = np.array([0, 0, 1]) # 正确解标签
>>> net.loss(x, t)
0.92806853663411326
```

接下来求梯度。和前面一样，我们使用numerical_gradient(f, x)求梯度（这里定义的函数f(W)的参数W是一个伪参数。因为numerical_gradient(f, x)会在内部执行f(x)，为了与之兼容而定义了f(W)）。

```
>>> def f(W):
...     return net.loss(x, t)
...
>>> dW = numerical_gradient(f, net.W)
>>> print(dW)
[[ 0.21924763  0.14356247 -0.36281009]
 [ 0.32887144  0.2153437  -0.54421514]]
```

numerical_gradient(f, x) 的参数f是函数，x是传给函数f的参数。因此，这里参数x取net.W，并定义一个计算损失函数的新函数f，然后把这个新定义的函数传递给numerical_gradient(f, x)。

numerical_gradient(f, net.W)的结果是dW，一个形状为2×3的二维数组。观察一下dW的内容，例如，会发现$\frac{\partial L}{\partial \boldsymbol{W}}$中的$\frac{\partial L}{\partial w_{11}}$的值大约是0.2，这表示如果将$w_{11}$增加$h$，那么损失函数的值会增加$0.2h$。再如，$\frac{\partial L}{\partial w_{23}}$对应的值大约是$-0.5$，这表示如果将$w_{23}$增加$h$，损失函数的值将减小$0.5h$。因此，从减小损失函数值的观点来看，$w_{23}$应向正方向更新，$w_{11}$应向负方向更新。至于更新的程度，$w_{23}$比$w_{11}$的贡献要大。

另外，在上面的代码中，定义新函数时使用了“def f(x):…”的形式。实际上，Python中如果定义的是简单的函数，可以使用lambda表示法。使用lambda的情况下，上述代码可以如下实现。

```
>>> f = lambda w: net.loss(x, t)
>>> dW = numerical_gradient(f, net.W)
```

求出神经网络的梯度后，接下来只需根据梯度法，更新权重参数即可。在下一节中，我们会以2层神经网络为例，实现整个学习过程。

> 为了对应形状为多维数组的权重参数W，这里使用的numerical_gradient()和之前的实现稍有不同。不过，改动只是为了对应多维数组，所以改动并不大。这里省略了对代码的说明，想知道细节的读者请参考源代码（common/gradient.py）。

4.5 学习算法的实现

关于神经网络学习的基础知识，到这里就全部介绍完了。“损失函数”“mini-batch”“梯度”“梯度下降法”等关键词已经陆续登场，这里我们来确认一下神经网络的学习步骤，顺便复习一下这些内容。神经网络的学习步骤如下所示。

前提

神经网络存在合适的权重和偏置，调整权重和偏置以便拟合训练数据的过程称为“学习”。神经网络的学习分成下面4个步骤。

步骤1（mini-batch）

从训练数据中随机选出一部分数据，这部分数据称为mini-batch。我们的目标是减小mini-batch的损失函数的值。

步骤2（计算梯度）

为了减小mini-batch的损失函数的值，需要求出各个权重参数的梯度。梯度表示损失函数的值减小最多的方向。

步骤3（更新参数）

将权重参数沿梯度方向进行微小更新。

步骤4(重复)

重复步骤1、步骤2、步骤3。

神经网络的学习按照上面4个步骤进行。这个方法通过梯度下降法更新参数，不过因为这里使用的数据是随机选择的mini batch数据，所以又称为**随机梯度下降法**(stochastic gradient descent)。“随机”指的是“随机选择的”的意思，因此，随机梯度下降法是“对随机选择的数据进行的梯度下降法”。深度学习的很多框架中，随机梯度下降法一般由一个名为**SGD**的函数来实现。SGD来源于随机梯度下降法的英文名称的首字母。

下面，我们来实现手写数字识别的神经网络。这里以2层神经网络(隐藏层为1层的网络)为对象，使用MNIST数据集进行学习。

4.5.1 2层神经网络的类

首先，我们将这个2层神经网络实现为一个名为`TwoLayerNet`的类，实现过程如下所示[1]。源代码在`ch04/two_layer_net.py`中。

```
import sys, os
sys.path.append(os.pardir)
from common.functions import *
from common.gradient import numerical_gradient

class TwoLayerNet:

    def __init__(self, input_size, hidden_size, output_size,
                 weight_init_std=0.01):
        # 初始化权重
        self.params = {}
        self.params['W1'] = weight_init_std * \
                            np.random.randn(input_size, hidden_size)
        self.params['b1'] = np.zeros(hidden_size)
        self.params['W2'] = weight_init_std * \
                            np.random.randn(hidden_size, output_size)
        self.params['b2'] = np.zeros(output_size)
```

① `TwoLayerNet`的实现参考了斯坦福大学CS231n课程提供的Python源代码。

```
    def predict(self, x):
        W1, W2 = self.params['W1'], self.params['W2']
        b1, b2 = self.params['b1'], self.params['b2']

        a1 = np.dot(x, W1) + b1
        z1 = sigmoid(a1)
        a2 = np.dot(z1, W2) + b2
        y = softmax(a2)

        return y

    # x:输入数据, t:监督数据
    def loss(self, x, t):
        y = self.predict(x)

        return cross_entropy_error(y, t)

    def accuracy(self, x, t):
        y = self.predict(x)
        y = np.argmax(y, axis=1)
        t = np.argmax(t, axis=1)

        accuracy = np.sum(y == t) / float(x.shape[0])
        return accuracy

    # x:输入数据, t:监督数据
    def numerical_gradient(self, x, t):
        loss_W = lambda W: self.loss(x, t)

        grads = {}
        grads['W1'] = numerical_gradient(loss_W, self.params['W1'])
        grads['b1'] = numerical_gradient(loss_W, self.params['b1'])
        grads['W2'] = numerical_gradient(loss_W, self.params['W2'])
        grads['b2'] = numerical_gradient(loss_W, self.params['b2'])

        return grads
```

虽然这个类的实现稍微有点长，但是因为和上一章的神经网络的前向处理的实现有许多共通之处，所以并没有太多新东西。我们先把这个类中用到的变量和方法整理一下。表4-1中只罗列了重要的变量，表4-2中则罗列了所有的方法。

表4-1 TwolayerNet类中使用的变量

变量	说明
params	保存神经网络的参数的字典型变量（实例变量）。 params['W1']是第1层的权重，params['b1']是第1层的偏置。 params['W2']是第2层的权重，params['b2']是第2层的偏置
grads	保存梯度的字典型变量（numerical_gradient()方法的返回值）。 grads['W1']是第1层权重的梯度，grads['b1']是第1层偏置的梯度。 grads['W2']是第2层权重的梯度，grads['b2']是第2层偏置的梯度

表4-2 TwoLayerNet类的方法

方法	说明
__init__(self, input_size, hidden_size, output_size)	进行初始化。 参数从头开始依次表示输入层的神经元数、隐藏层的神经元数、输出层的神经元数
predict(self, x)	进行识别（推理）。 参数x是图像数据
loss(self, x, t)	计算损失函数的值。 参数x是图像数据，t是正确解标签（后面3个方法的参数也一样）
accuracy(self, x, t)	计算识别精度
numerical_gradient(self, x, t)	计算权重参数的梯度
gradient(self, x, t)	计算权重参数的梯度。 numerical_gradient()的高速版，将在下一章实现

TwoLayerNet类有params和grads两个字典型实例变量。params变量中保存了权重参数，比如params['W1']以NumPy数组的形式保存了第1层的权重参数。此外，第1层的偏置可以通过param['b1']进行访问。这里来看一个例子。

```
net = TwoLayerNet(input_size=784, hidden_size=100, output_size=10)
net.params['W1'].shape # (784, 100)
net.params['b1'].shape # (100,)
net.params['W2'].shape # (100, 10)
net.params['b2'].shape # (10,)
```

如上所示，params变量中保存了该神经网络所需的全部参数。并且，params变量中保存的权重参数会用在推理处理（前向处理）中。顺便说一下，推理处理的实现如下所示。

```
x = np.random.rand(100, 784) # 伪输入数据（100笔）
y = net.predict(x)
```

此外，与params变量对应，grads变量中保存了各个参数的梯度。如下所示，使用numerical_gradient()方法计算梯度后，梯度的信息将保存在grads变量中。

```
x = np.random.rand(100, 784) # 伪输入数据（100笔）
t = np.random.rand(100, 10)  # 伪正确解标签（100笔）

grads = net.numerical_gradient(x, t)  # 计算梯度

grads['W1'].shape  # (784, 100)
grads['b1'].shape  # (100,)
grads['W2'].shape  # (100, 10)
grads['b2'].shape  # (10,)
```

接着，我们来看一下TwoLayerNet的方法的实现。首先是__init__(self, input_size, hidden_size, output_size)方法，它是类的初始化方法（所谓初始化方法，就是生成TwoLayerNet实例时被调用的方法）。从第1个参数开始，依次表示输入层的神经元数、隐藏层的神经元数、输出层的神经元数。另外，因为进行手写数字识别时，输入图像的大小是784（28×28），输出为10个类别，所以指定参数input_size=784、output_size=10，将隐藏层的个数hidden_size设置为一个合适的值即可。

此外，这个初始化方法会对权重参数进行初始化。如何设置权重参数的初始值这个问题是关系到神经网络能否成功学习的重要问题。后面我们会详细讨论权重参数的初始化，这里只需要知道，权重使用符合高斯分布的随机数进行初始化，偏置使用0进行初始化。predict(self, x)和accuracy(self, x, t)的实现和上一章的神经网络的推理处理基本一样。如果仍有不明白的地方，请再回顾一下上一章的内容。另外，loss(self, x, t)

是计算损失函数值的方法。这个方法会基于predict()的结果和正确解标签，计算交叉熵误差。

剩下的numerical_gradient(self, x, t)方法会计算各个参数的梯度。根据数值微分，计算各个参数相对于损失函数的梯度。另外，gradient(self, x, t)是下一章要实现的方法，该方法使用误差反向传播法高效地计算梯度。

> numerical_gradient(self, x, t)基于数值微分计算参数的梯度。下一章，我们会介绍一个高速计算梯度的方法，称为误差反向传播法。用误差反向传播法求到的梯度和数值微分的结果基本一致，但可以高速地进行处理。使用误差反向传播法计算梯度的gradient(self, x, t)方法会在下一章实现，不过考虑到神经网络的学习比较花时间，想节约学习时间的读者可以替换掉这里的numerical_gradient(self, x, t)，抢先使用gradient(self, x, t)！

4.5.2 mini-batch的实现

神经网络的学习的实现使用的是前面介绍过的mini-batch学习。所谓mini-batch学习，就是从训练数据中随机选择一部分数据（称为mini-batch），再以这些mini-batch为对象，使用梯度法更新参数的过程。下面，我们就以TwoLayerNet类为对象，使用MNIST数据集进行学习（源代码在ch04/train_neuralnet.py中）。

```
import numpy as np
from dataset.mnist import load_mnist
from two_layer_net import TwoLayerNet

(x_train, t_train), (x_test, t_test) = \ load_mnist(normalize=True, one_hot_
laobel = True)

train_loss_list = []

# 超参数
iters_num = 10000
train_size = x_train.shape[0]
batch_size = 100
learning_rate = 0.1
```

```
network = TwoLayerNet(input_size=784, hidden_size=50, output_size=10)

for i in range(iters_num):
    # 获取mini-batch
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    t_batch = t_train[batch_mask]

    # 计算梯度
    grad = network.numerical_gradient(x_batch, t_batch)
    # grad = network.gradient(x_batch, t_batch) # 高速版!

    # 更新参数
    for key in ('W1', 'b1', 'W2', 'b2'):
        network.params[key] -= learning_rate * grad[key]

    # 记录学习过程
    loss = network.loss(x_batch, t_batch)
    train_loss_list.append(loss)
```

这里，mini-batch的大小为100，需要每次从60000个训练数据中随机取出100个数据（图像数据和正确解标签数据）。然后，对这个包含100笔数据的mini-batch求梯度，使用随机梯度下降法（SGD）更新参数。这里，梯度法的更新次数（循环的次数）为10000。每更新一次，都对训练数据计算损失函数的值，并把该值添加到数组中。用图像来表示这个损失函数的值的推移，如图4-11所示。

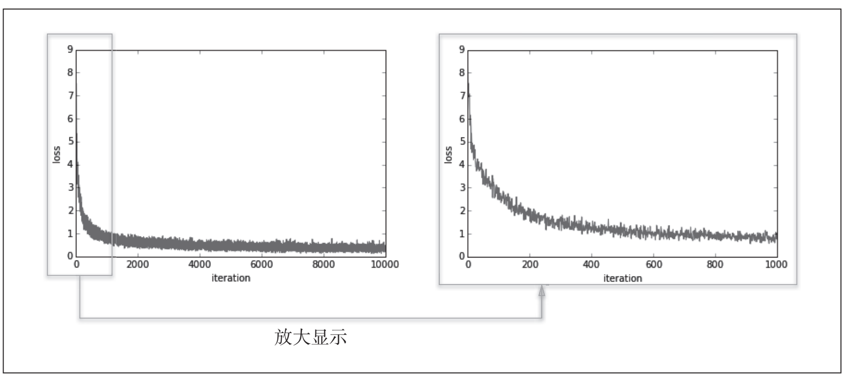


图4-11　损失函数的推移：左图是10000次循环的推移，右图是1000次循环的推移

观察图4-11，可以发现随着学习的进行，损失函数的值在不断减小。这是学习正常进行的信号，表示神经网络的权重参数在逐渐拟合数据。也就是说，神经网络的确在学习！通过反复地向它浇灌（输入）数据，神经网络正在逐渐向最优参数靠近。

4.5.3 基于测试数据的评价

根据图4-11呈现的结果，我们确认了通过反复学习可以使损失函数的值逐渐减小这一事实。不过这个损失函数的值，严格地讲是“对训练数据的某个mini-batch的损失函数”的值。训练数据的损失函数值减小，虽说是神经网络的学习正常进行的一个信号，但光看这个结果还不能说明该神经网络在其他数据集上也一定能有同等程度的表现。

神经网络的学习中，必须确认是否能够正确识别训练数据以外的其他数据，即确认是否会发生过拟合。过拟合是指，虽然训练数据中的数字图像能被正确辨别，但是不在训练数据中的数字图像却无法被识别的现象。

神经网络学习的最初目标是掌握泛化能力，因此，要评价神经网络的泛化能力，就必须使用不包含在训练数据中的数据。下面的代码在进行学习的过程中，会定期地对训练数据和测试数据记录识别精度。这里，每经过一个epoch，我们都会记录下训练数据和测试数据的识别精度。

epoch是一个单位。一个epoch表示学习中所有训练数据均被使用过一次时的更新次数。比如，对于10000笔训练数据，用大小为100笔数据的mini-batch进行学习时，重复随机梯度下降法100次，所有的训练数据就都被“看过”了[①]。此时，100次就是一个epoch。

为了正确进行评价，我们来稍稍修改一下前面的代码。与前面的代码不同的地方，我们用粗体来表示。

---

① 实际上，一般做法是事先将所有训练数据随机打乱，然后按指定的批次大小，按序生成mini-batch。这样每个mini-batch均有一个索引号，比如此例可以是$0, 1, 2, \cdots, 99$，然后用索引号可以遍历所有的mini-batch。遍历一次所有数据，就称为一个epoch。请注意，本节中的mini-batch每次都是随机选择的，所以不一定每个数据都会被看到。——译者注

```
import numpy as np
from dataset.mnist import load_mnist
from two_layer_net import TwoLayerNet

(x_train, t_train), (x_test, t_test) = \ load_mnist(normalize=True, one_hot_
laobel = True)

train_loss_list = []
train_acc_list = []
test_acc_list = []
# 平均每个epoch的重复次数
iter_per_epoch = max(train_size / batch_size, 1)

# 超参数
iters_num = 10000
batch_size = 100
learning_rate = 0.1

network = TwoLayerNet(input_size=784, hidden_size=50, output_size=10)

for i in range(iters_num):
    # 获取mini-batch
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    t_batch = t_train[batch_mask]

    # 计算梯度
    grad = network.numerical_gradient(x_batch, t_batch)
    # grad = network.gradient(x_batch, t_batch) # 高速版!

    # 更新参数
    for key in ('W1', 'b1', 'W2', 'b2'):
        network.params[key] -= learning_rate * grad[key]

    loss = network.loss(x_batch, t_batch)
    train_loss_list.append(loss)
    # 计算每个epoch的识别精度
    if i % iter_per_epoch == 0:
        train_acc = network.accuracy(x_train, t_train)
        test_acc = network.accuracy(x_test, t_test)
        train_acc_list.append(train_acc)
        test_acc_list.append(test_acc)
        print("train acc, test acc | " + str(train_acc) + ", " + str(test_acc))
```

在上面的例子中，每经过一个epoch，就对所有的训练数据和测试数据计算识别精度，并记录结果。之所以要计算每一个epoch的识别精度，是因为如果在`for`语句的循环中一直计算识别精度，会花费太多时间。并且，也

没有必要那么频繁地记录识别精度（只要从大方向上大致把握识别精度的推移就可以了）。因此，我们才会每经过一个epoch就记录一次训练数据的识别精度。

把从上面的代码中得到的结果用图表示的话，如图4-12所示。

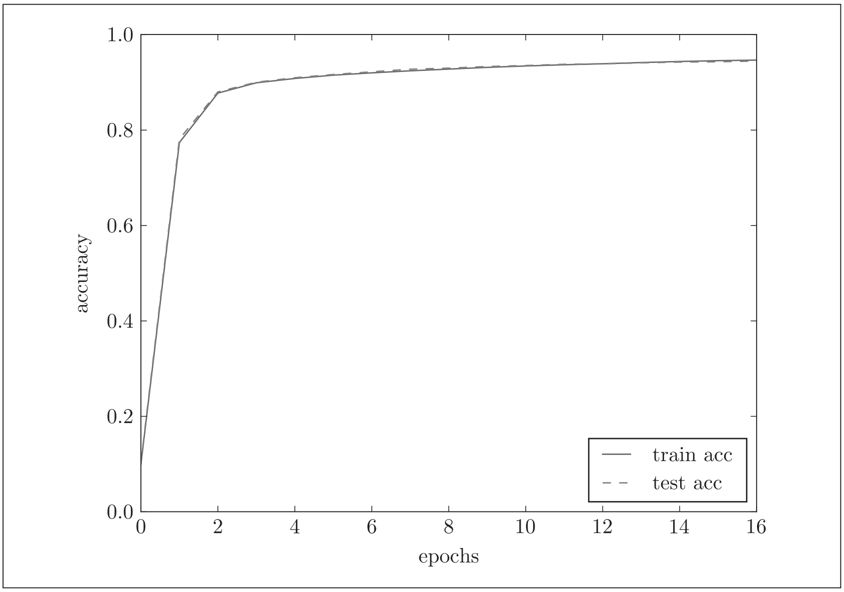


图4-12　训练数据和测试数据的识别精度的推移（横轴的单位是epoch）

图4-12中，实线表示训练数据的识别精度，虚线表示测试数据的识别精度。如图所示，随着epoch的前进（学习的进行），我们发现使用训练数据和测试数据评价的识别精度都提高了，并且，这两个识别精度基本上没有差异（两条线基本重叠在一起）。因此，可以说这次的学习中没有发生过拟合的现象。

4.6　小结

本章中，我们介绍了神经网络的学习。首先，为了能顺利进行神经网络的学习，我们导入了损失函数这个指标。以这个损失函数为基准，找出使它

的值达到最小的权重参数，就是神经网络学习的目标。为了找到尽可能小的损失函数值，我们介绍了使用函数斜率的梯度法。

本章所学的内容

- 机器学习中使用的数据集分为训练数据和测试数据。
- 神经网络用训练数据进行学习，并用测试数据评价学习到的模型的泛化能力。
- 神经网络的学习以损失函数为指标，更新权重参数，以使损失函数的值减小。
- 利用某个给定的微小值的差分求导数的过程，称为数值微分。
- 利用数值微分，可以计算权重参数的梯度。
- 数值微分虽然费时间，但是实现起来很简单。下一章中要实现的稍微复杂一些的误差反向传播法可以高速地计算梯度。

第5章 误差反向传播法

上一章中，我们介绍了神经网络的学习，并通过数值微分计算了神经网络的权重参数的梯度（严格来说，是损失函数关于权重参数的梯度）。数值微分虽然简单，也容易实现，但缺点是计算上比较费时间。本章我们将学习一个能够高效计算权重参数的梯度的方法——误差反向传播法。

要正确理解误差反向传播法，我个人认为有两种方法：一种是基于数学式；另一种是基于**计算图**（computational graph）。前者是比较常见的方法，机器学习相关的图书中多数都是以数学式为中心展开论述的。因为这种方法严密且简洁，所以确实非常合理，但如果一上来就围绕数学式进行探讨，会忽略一些根本的东西，止步于式子的罗列。因此，本章希望大家通过计算图，直观地理解误差反向传播法。然后，再结合实际的代码加深理解，相信大家一定会有种“原来如此！”的感觉。

此外，通过计算图来理解误差反向传播法这个想法，参考了Andrej Karpathy的博客[4]和他与Fei-Fei Li教授负责的斯坦福大学的深度学习课程CS231n[5]。

5.1 计算图

计算图将计算过程用图形表示出来。这里说的图形是数据结构图，通过多个节点和边表示（连接节点的直线称为“边”）。为了让大家熟悉计算图，

本节先用计算图解一些简单的问题。从这些简单的问题开始，逐步深入，最终抵达误差反向传播法。

5.1.1 用计算图求解

现在，我们尝试用计算图解简单的问题。下面我们要看的几个问题都是用心算就能解开的简单问题，这里的目的只是通过它们让大家熟悉计算图。掌握了计算图的使用方法之后，在后面即将看到的复杂计算中它将发挥巨大威力，所以本节请一定学会计算图的使用方法。

问题1：太郎在超市买了2个100日元一个的苹果，消费税是10%，请计算支付金额。

计算图通过节点和箭头表示计算过程。节点用○表示，○中是计算的内容。将计算的中间结果写在箭头的上方，表示各个节点的计算结果从左向右传递。用计算图解问题1，求解过程如图5-1所示。

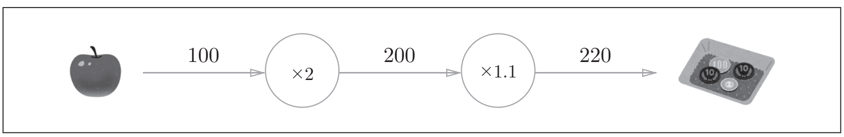


图5-1 基于计算图求解的问题1的答案

如图5-1所示，开始时，苹果的100日元流到“× 2”节点，变成200日元，然后被传递给下一个节点。接着，这个200日元流向“× 1.1”节点，变成220日元。因此，从这个计算图的结果可知，答案为220日元。

虽然图5-1中把“× 2”“× 1.1”等作为一个运算整体用○括起来了，不过只用○表示乘法运算“×”也是可行的。此时，如图5-2所示，可以将“2”和“1.1”分别作为变量“苹果的个数”和“消费税”标在○外面。

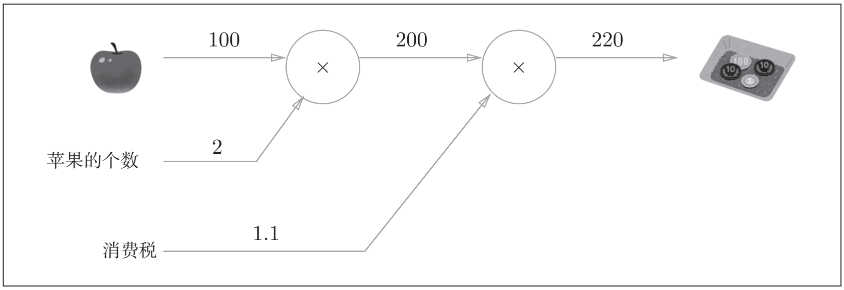


图5-2　基于计算图求解的问题1的答案:“苹果的个数”和“消费税”作为变量标在○外面

再看下一题。

问题2：太郎在超市买了2个苹果、3个橘子。其中，苹果每个100日元，橘子每个150日元。消费税是10%，请计算支付金额。

同问题1，我们用计算图来解问题2，求解过程如图5-3所示。

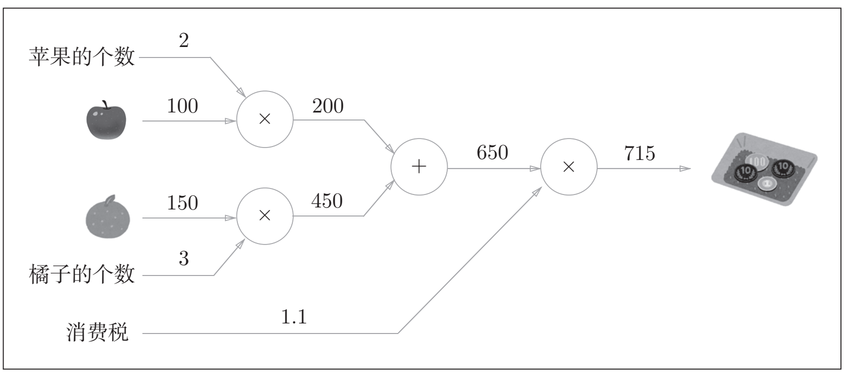


图5-3　基于计算图求解的问题2的答案

这个问题中新增了加法节点“+”，用来合计苹果和橘子的金额。构建了计算图后，从左向右进行计算。就像电路中的电流流动一样，计算结果从左向右传递。到达最右边的计算结果后，计算过程就结束了。从图5-3中可知，问题2的答案为715日元。

综上，用计算图解题的情况下，需要按如下流程进行。

1. 构建计算图。
2. 在计算图上，从左向右进行计算。

这里的第2步“从左向右进行计算”是一种正方向上的传播，简称为**正向传播**（forward propagation）。正向传播是从计算图出发点到结束点的传播。既然有正向传播这个名称，当然也可以考虑反向（从图上看的话，就是从右向左）的传播。实际上，这种传播称为**反向传播**（backward propagation）。反向传播将在接下来的导数计算中发挥重要作用。

5.1.2 局部计算

计算图的特征是可以通过传递“局部计算”获得最终结果。“局部”这个词的意思是“与自己相关的某个小范围”。局部计算是指，无论全局发生了什么，都能只根据与自己相关的信息输出接下来的结果。

我们用一个具体的例子来说明局部计算。比如，在超市买了2个苹果和其他很多东西。此时，可以画出如图5-4所示的计算图。

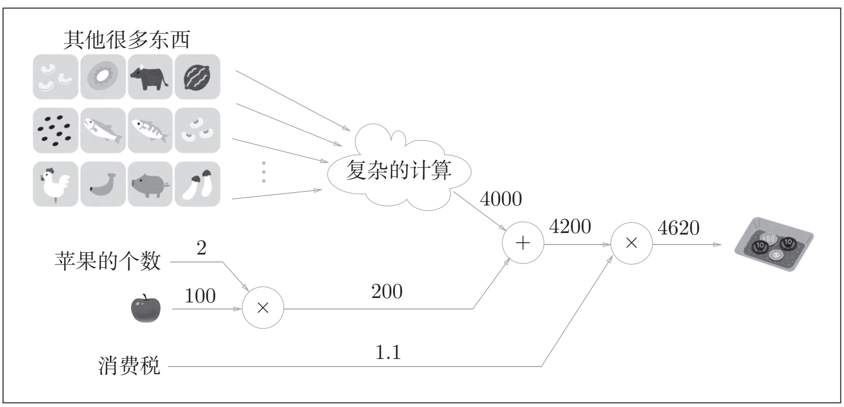


图5-4 买了2个苹果和其他很多东西的例子

如图5-4所示，假设（经过复杂的计算）购买的其他很多东西总共花费

4000日元。这里的重点是，各个节点处的计算都是局部计算。这意味着，例如苹果和其他很多东西的求和运算（$4000 + 200 \rightarrow 4200$）并不关心4000这个数字是如何计算而来的，只要把两个数字相加就可以了。换言之，各个节点处只需进行与自己有关的计算（在这个例子中是对输入的两个数字进行加法运算），不用考虑全局。

综上，计算图可以集中精力于局部计算。无论全局的计算有多么复杂，各个步骤所要做的就是对象节点的局部计算。虽然局部计算非常简单，但是通过传递它的计算结果，可以获得全局的复杂计算的结果。

比如，组装汽车是一个复杂的工作，通常需要进行“流水线”作业。每个工人（机器）所承担的都是被简化了的工作，这个工作的成果会传递给下一个工人，直至汽车组装完成。计算图将复杂的计算分割成简单的局部计算，和流水线作业一样，将局部计算的结果传递给下一个节点。在将复杂的计算分解成简单的计算这一点上与汽车的组装有相似之处。

5.1.3 为何用计算图解题

前面我们用计算图解答了两个问题，那么计算图到底有什么优点呢？一个优点就在于前面所说的局部计算。无论全局是多么复杂的计算，都可以通过局部计算使各个节点致力于简单的计算，从而简化问题。另一个优点是，利用计算图可以将中间的计算结果全部保存起来（比如，计算进行到2个苹果时的金额是200日元、加上消费税之前的金额650日元等）。但是只有这些理由可能还无法令人信服。实际上，使用计算图最大的原因是，可以通过反向传播高效计算导数。

在介绍计算图的反向传播时，我们再来思考一下问题1。问题1中，我们计算了购买2个苹果时加上消费税最终需要支付的金额。这里，假设我们想知道苹果价格的上涨会在多大程度上影响最终的支付金额，即求“支付金额关于苹果的价格的导数”。设苹果的价格为x，支付金额为L，则相当于求$\frac{\partial L}{\partial x}$。这个导数的值表示当苹果的价格稍微上涨时，支付金额会增加多少。

如前所述，“支付金额关于苹果的价格的导数”的值可以通过计算图的反向传播求出来。先来看一下结果，如图5-5所示，可以通过计算图的反向传播求导数（关于如何进行反向传播，接下来马上会介绍）。

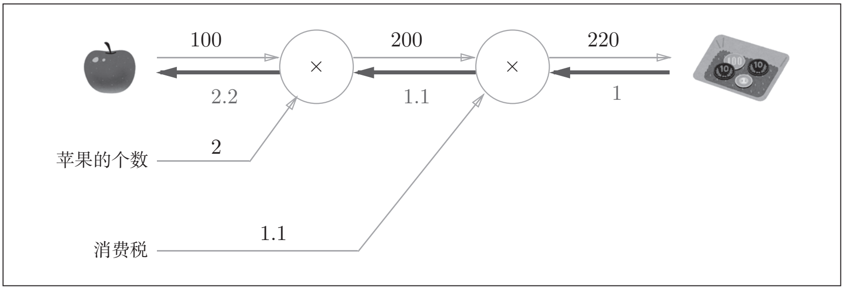


图5-5　基于反向传播的导数的传递

如图5-5所示，反向传播使用与正方向相反的箭头（粗线）表示。反向传播传递“局部导数”，将导数的值写在箭头的下方。在这个例子中，反向传播从右向左传递导数的值（$1 \rightarrow 1.1 \rightarrow 2.2$）。从这个结果中可知，“支付金额关于苹果的价格的导数”的值是2.2。这意味着，如果苹果的价格上涨1日元，最终的支付金额会增加2.2日元（严格地讲，如果苹果的价格增加某个微小值，则最终的支付金额将增加那个微小值的2.2倍）。

这里只求了关于苹果的价格的导数，不过“支付金额关于消费税的导数”“支付金额关于苹果的个数的导数”等也都可以用同样的方式算出来。并且，计算中途求得的导数的结果（中间传递的导数）可以被共享，从而可以高效地计算多个导数。综上，计算图的优点是，可以通过正向传播和反向传播高效地计算各个变量的导数值。

5.2　链式法则

前面介绍的计算图的正向传播将计算结果正向（从左到右）传递，其计算过程是我们日常接触的计算过程，所以感觉上可能比较自然。而反向传播

将局部导数向正方向的反方向（从右到左）传递，一开始可能会让人感到困惑。传递这个局部导数的原理，是基于**链式法则**（chain rule）的。本节将介绍链式法则，并阐明它是如何对应计算图上的反向传播的。

5.2.1 计算图的反向传播

话不多说，让我们先来看一个使用计算图的反向传播的例子。假设存在 $y = f(x)$ 的计算，这个计算的反向传播如图 5-6 所示。

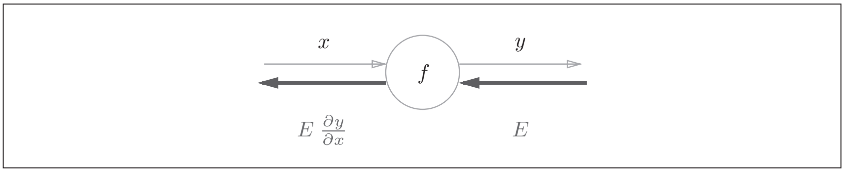


图 5-6 计算图的反向传播：沿着与正方向相反的方向，乘上局部导数

如图所示，反向传播的计算顺序是，将信号 E 乘以节点的局部导数（$\frac{\partial y}{\partial x}$），然后将结果传递给下一个节点。这里所说的局部导数是指正向传播中 $y = f(x)$ 的导数，也就是 y 关于 x 的导数（$\frac{\partial y}{\partial x}$）。比如，假设 $y = f(x) = x^2$，则局部导数为 $\frac{\partial y}{\partial x} = 2x$。把这个局部导数乘以上游传过来的值（本例中为 E），然后传递给前面的节点。

这就是反向传播的计算顺序。通过这样的计算，可以高效地求出导数的值，这是反向传播的要点。那么这是如何实现的呢？我们可以从链式法则的原理进行解释。下面我们就来介绍链式法则。

5.2.2 什么是链式法则

介绍链式法则时，我们需要先从**复合函数**说起。复合函数是由多个函数构成的函数。比如，$z = (x + y)^2$ 是由式（5.1）所示的两个式子构成的。

$$\begin{aligned} z &= t^2 \\ t &= x + y \end{aligned} \tag{5.1}$$

链式法则是关于复合函数的导数的性质，定义如下。

> 如果某个函数由复合函数表示，则该复合函数的导数可以用构成复合函数的各个函数的导数的乘积表示。

这就是链式法则的原理，乍一看可能比较难理解，但实际上它是一个非常简单的性质。以式(5.1)为例，$\frac{\partial z}{\partial x}$($z$关于$x$的导数)可以用$\frac{\partial z}{\partial t}$($z$关于$t$的导数)和$\frac{\partial t}{\partial x}$($t$关于$x$的导数)的乘积表示。用数学式表示的话，可以写成式(5.2)。

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial t}\frac{\partial t}{\partial x} \tag{5.2}$$

式(5.2)中的∂t正好可以像下面这样"互相抵消"，所以记起来很简单。

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\cancel{\partial t}}\frac{\cancel{\partial t}}{\partial x}$$

现在我们使用链式法则，试着求式(5.2)的导数$\frac{\partial z}{\partial x}$。为此，我们要先求式(5.1)中的局部导数(偏导数)。

$$\begin{aligned} \frac{\partial z}{\partial t} &= 2t \\ \frac{\partial t}{\partial x} &= 1 \end{aligned} \tag{5.3}$$

如式(5.3)所示，$\frac{\partial z}{\partial t}$等于$2t$，$\frac{\partial t}{\partial x}$等于1。这是基于导数公式的解析解。然后，最后要计算的$\frac{\partial z}{\partial x}$可由式(5.3)求得的导数的乘积计算出来。

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial t}\frac{\partial t}{\partial x} = 2t \cdot 1 = 2(x+y) \tag{5.4}$$

5.2.3 链式法则和计算图

现在我们尝试将式(5.4)的链式法则的计算用计算图表示出来。如果用"**2"节点表示平方运算的话，则计算图如图5-7所示。

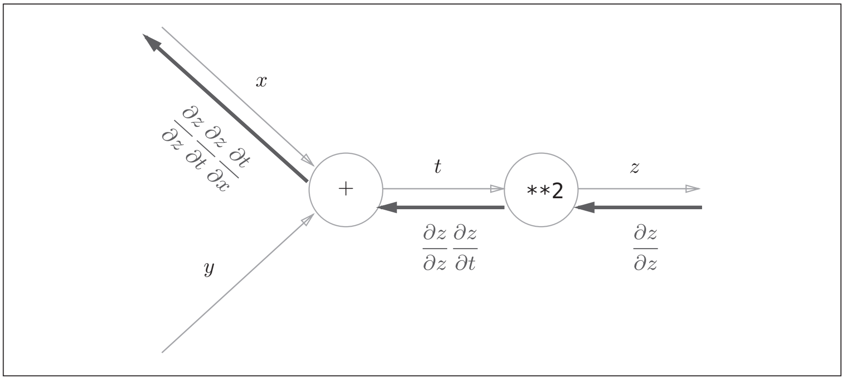


图5-7　式(5.4)的计算图：沿着与正方向相反的方向，乘上局部导数后传递

如图所示，计算图的反向传播从右到左传播信号。反向传播的计算顺序是，先将节点的输入信号乘以节点的局部导数(偏导数)，然后再传递给下一个节点。比如，反向传播时，"**2"节点的输入是 $\frac{\partial z}{\partial x}$，将其乘以局部导数 $\frac{\partial z}{\partial t}$(因为正向传播时输入是 t、输出是 z，所以这个节点的局部导数是 $\frac{\partial z}{\partial t}$)，然后传递给下一个节点。另外，图5-7中反向传播最开始的信号 $\frac{\partial z}{\partial z}$ 在前面的数学式中没有出现，这是因为 $\frac{\partial z}{\partial z}=1$，所以在刚才的式子中被省略了。

图5-7中需要注意的是最左边的反向传播的结果。根据链式法则，$\frac{\partial z}{\partial z}\frac{\partial z}{\partial t}\frac{\partial t}{\partial x}=\frac{\partial z}{\partial t}\frac{\partial t}{\partial x}=\frac{\partial z}{\partial x}$ 成立，对应"z关于x的导数"。也就是说，反向传播是基于链式法则的。

把式(5.3)的结果代入到图5-7中，结果如图5-8所示，$\frac{\partial z}{\partial x}$ 的结果为 $2(x+y)$。

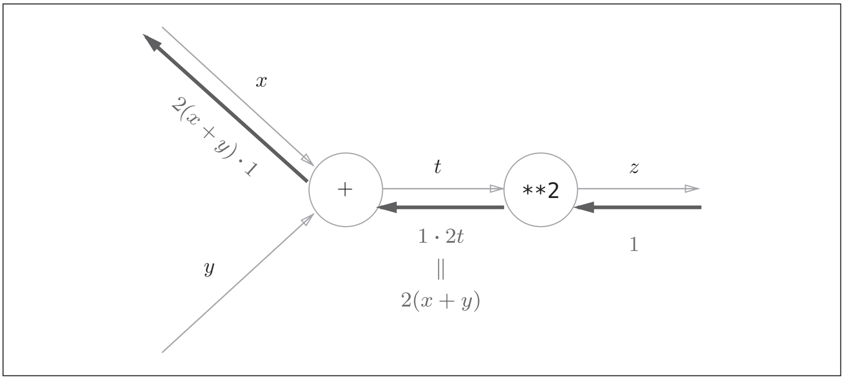


图5-8 根据计算图的反向传播的结果，$\frac{\partial z}{\partial x}$ 等于 $2(x+y)$

5.3 反向传播

上一节介绍了计算图的反向传播是基于链式法则成立的。本节将以“+”和“×”等运算为例，介绍反向传播的结构。

5.3.1 加法节点的反向传播

首先来考虑加法节点的反向传播。这里以$z = x + y$为对象，观察它的反向传播。$z = x + y$的导数可由下式（解析性地）计算出来。

$$\frac{\partial z}{\partial x} = 1 \\ \frac{\partial z}{\partial y} = 1 \tag{5.5}$$

如式(5.5)所示，$\frac{\partial z}{\partial x}$ 和 $\frac{\partial z}{\partial y}$ 同时都等于1。因此，用计算图表示的话，如图5-9所示。

在图5-9中，反向传播将从上游传过来的导数（本例中是 $\frac{\partial L}{\partial z}$）乘以1，然后传向下游。也就是说，因为加法节点的反向传播只乘以1，所以输入的值会原封不动地流向下一个节点。

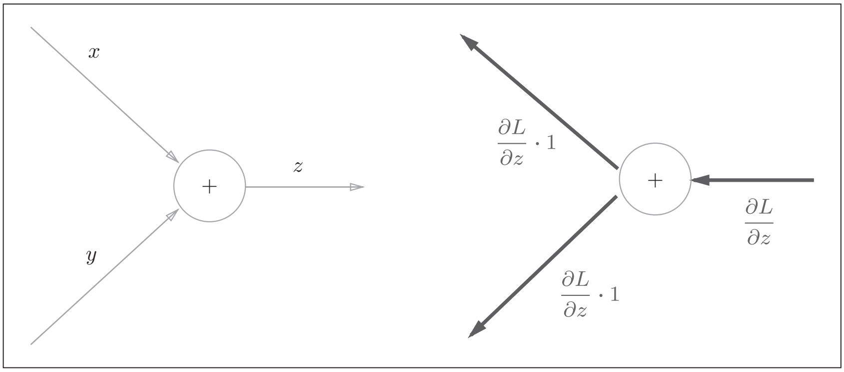


图 5-9　加法节点的反向传播：左图是正向传播，右图是反向传播。如右图的反向传播所示，加法节点的反向传播将上游的值原封不动地输出到下游

另外，本例中把从上游传过来的导数的值设为 $\frac{\partial L}{\partial z}$。这是因为，如图 5-10 所示，我们假定了一个最终输出值为 L 的大型计算图。$z = x + y$ 的计算位于这个大型计算图的某个地方，从上游会传来 $\frac{\partial L}{\partial z}$ 的值，并向下游传递 $\frac{\partial L}{\partial x}$ 和 $\frac{\partial L}{\partial y}$。

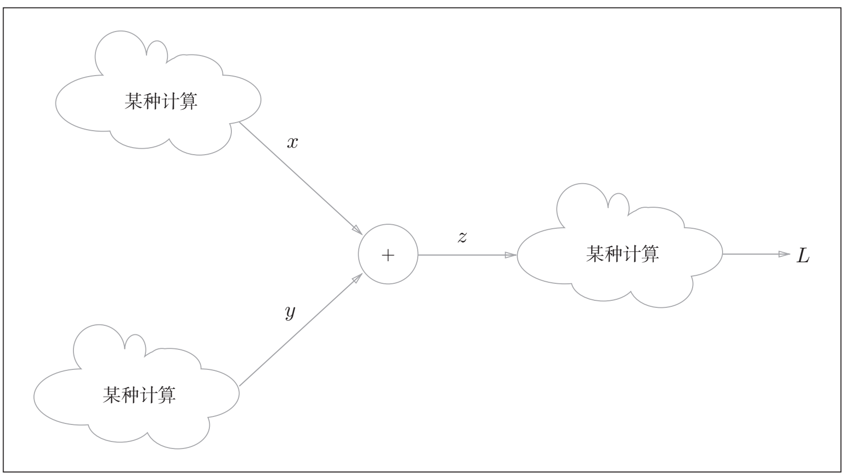


图 5-10　加法节点存在于某个最后输出的计算的一部分中。反向传播时，从最右边的输出出发，局部导数从节点向节点反方向传播

现在来看一个加法的反向传播的具体例子。假设有“10 + 5=15”这一计算，反向传播时，从上游会传来值1.3。用计算图表示的话，如图5-11所示。

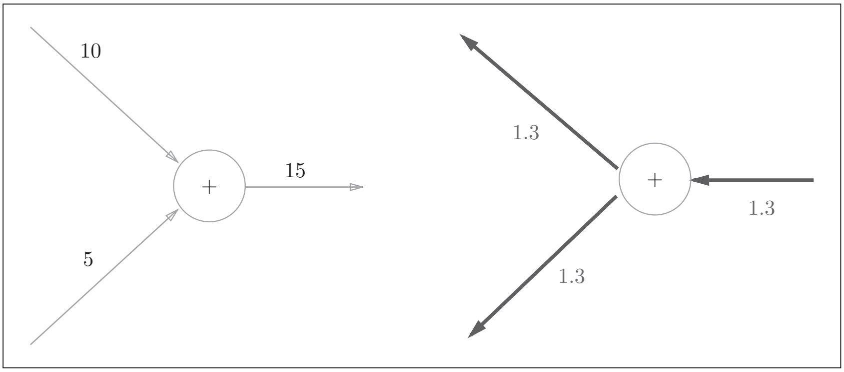


图5-11 加法节点的反向传播的具体例子

因为加法节点的反向传播只是将输入信号输出到下一个节点，所以如图5-11所示，反向传播将1.3向下一个节点传递。

5.3.2 乘法节点的反向传播

接下来，我们看一下乘法节点的反向传播。这里我们考虑$z = xy$。这个式子的导数用式(5.6)表示。

$$\begin{aligned}\frac{\partial z}{\partial x} &= y \\ \frac{\partial z}{\partial y} &= x\end{aligned} \tag{5.6}$$

根据式(5.6)，可以像图5-12那样画计算图。

乘法的反向传播会将上游的值乘以正向传播时的输入信号的“翻转值”后传递给下游。翻转值表示一种翻转关系，如图5-12所示，正向传播时信号是x的话，反向传播时则是y；正向传播时信号是y的话，反向传播时则是x。

现在我们来看一个具体的例子。比如，假设有“$10 \times 5 = 50$”这一计算，反向传播时，从上游会传来值1.3。用计算图表示的话，如图5-13所示。

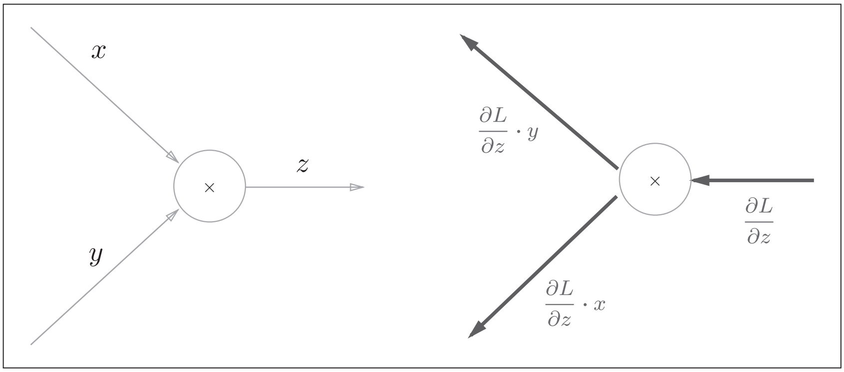


图5-12　乘法的反向传播：左图是正向传播，右图是反向传播

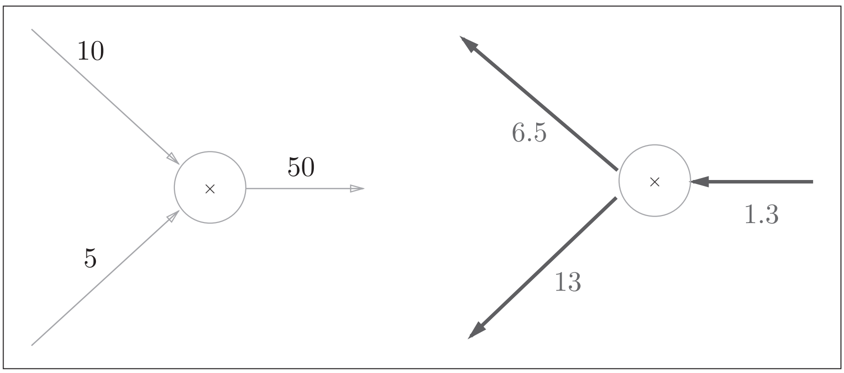


图5-13　乘法节点的反向传播的具体例子

因为乘法的反向传播会乘以输入信号的翻转值，所以各自可按$1.3 \times 5 = 6.5$、$1.3 \times 10 = 13$计算。另外，加法的反向传播只是将上游的值传给下游，并不需要正向传播的输入信号。但是，乘法的反向传播需要正向传播时的输入信号值。因此，实现乘法节点的反向传播时，要保存正向传播的输入信号。

5.3.3　苹果的例子

再来思考一下本章最开始举的购买苹果的例子（2个苹果和消费税）。这里要解的问题是苹果的价格、苹果的个数、消费税这3个变量各自如何影响最终支付的金额。这个问题相当于求“支付金额关于苹果的价格的导数”“支

付金额关于苹果的个数的导数”“支付金额关于消费税的导数”。用计算图的反向传播来解的话，求解过程如图5-14所示。

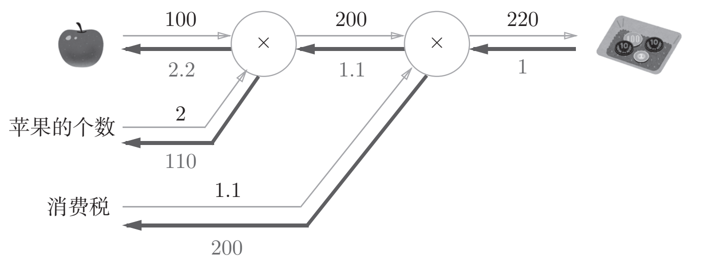


图5-14 购买苹果的反向传播的例子

如前所述，乘法节点的反向传播会将输入信号翻转后传给下游。从图5-14的结果可知，苹果的价格的导数是2.2，苹果的个数的导数是110，消费税的导数是200。这可以解释为，如果消费税和苹果的价格增加相同的值，则消费税将对最终价格产生200倍大小的影响，苹果的价格将产生2.2倍大小的影响。不过，因为这个例子中消费税和苹果的价格的量纲不同，所以才形成了这样的结果（消费税的1是100%，苹果的价格的1是1日元）。

最后作为练习，请大家来试着解一下“购买苹果和橘子”的反向传播。在图5-15中的方块中填入数字，求各个变量的导数（答案在若干页后）。

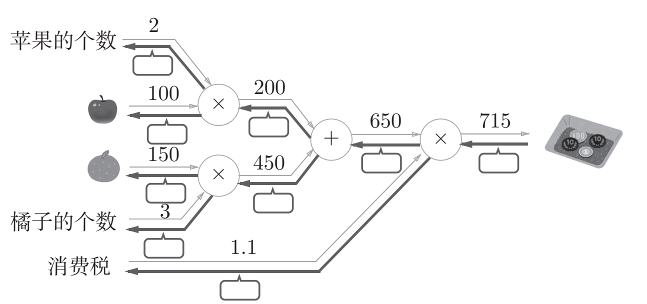


图5-15 购买苹果和橘子的反向传播的例子：在方块中填入数字，完成反向传播

5.4 简单层的实现

本节将用Python实现前面的购买苹果的例子。这里，我们把要实现的计算图的乘法节点称为“乘法层”（MulLayer），加法节点称为“加法层”（AddLayer）。

> 下一节，我们将把构建神经网络的“层”实现为一个类。这里所说的“层”是神经网络中功能的单位。比如，负责sigmoid函数的Sigmoid、负责矩阵乘积的Affine等，都以层为单位进行实现。因此，这里也以层为单位来实现乘法节点和加法节点。

5.4.1 乘法层的实现

层的实现中有两个共通的方法（接口）forward()和backward()。forward()对应正向传播，backward()对应反向传播。

现在来实现乘法层。乘法层作为MulLayer类，其实现过程如下所示（源代码在ch05/layer_naive.py中）。

```
class MulLayer:
    def __init__(self):
        self.x = None
        self.y = None

    def forward(self, x, y):
        self.x = x
        self.y = y
        out = x * y

        return out

    def backward(self, dout):
        dx = dout * self.y # 翻转x和y
        dy = dout * self.x

        return dx, dy
```

__init__()中会初始化实例变量x和y，它们用于保存正向传播时的输入值。forward()接收x和y两个参数，将它们相乘后输出。backward()将从上游传来的导数(dout)乘以正向传播的翻转值，然后传给下游。

上面就是MulLayer的实现。现在我们使用MulLayer实现前面的购买苹果的例子(2个苹果和消费税)。上一节中我们使用计算图的正向传播和反向传播，像图5-16这样进行了计算。

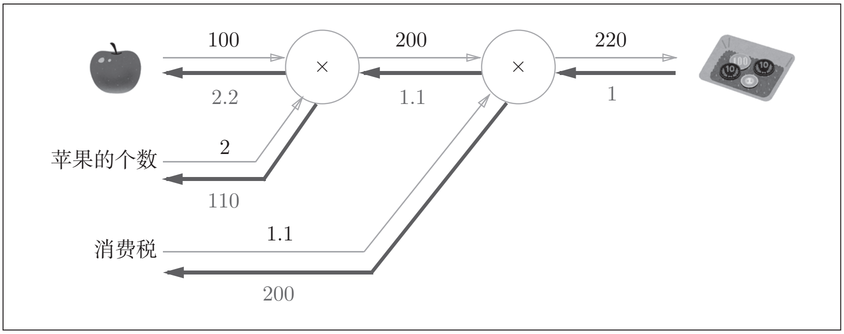


图5-16 购买2个苹果

使用这个乘法层的话，图5-16的正向传播可以像下面这样实现(源代码在ch05/buy_apple.py中)。

```
apple = 100
apple_num = 2
tax = 1.1

# layer
mul_apple_layer = MulLayer()
mul_tax_layer = MulLayer()

# forward
apple_price = mul_apple_layer.forward(apple, apple_num)
price = mul_tax_layer.forward(apple_price, tax)

print(price) # 220
```

此外，关于各个变量的导数可由backward()求出。

```
# backward
dprice = 1
dapple_price, dtax = mul_tax_layer.backward(dprice)
dapple, dapple_num = mul_apple_layer.backward(dapple_price)

print(dapple, dapple_num, dtax) # 2.2 110 200
```

这里，调用backward()的顺序与调用forward()的顺序相反。此外，要注意backward()的参数中需要输入“关于正向传播时的输出变量的导数”。比如，mul_apple_layer乘法层在正向传播时会输出apple_price，在反向传播时，则会将apple_price的导数dapple_price设为参数。另外，这个程序的运行结果和图5-16是一致的。

5.4.2 加法层的实现

接下来，我们实现加法节点的加法层，如下所示。

```
class AddLayer:
    def __init__(self):
        pass

    def forward(self, x, y):
        out = x + y
        return out

    def backward(self, dout):
        dx = dout * 1
        dy = dout * 1
        return dx, dy
```

加法层不需要特意进行初始化，所以__init__()中什么也不运行(pass语句表示“什么也不运行”)。加法层的forward()接收x和y两个参数，将它们相加后输出。backward()将上游传来的导数(dout)原封不动地传递给下游。

现在，我们使用加法层和乘法层，实现图5-17所示的购买2个苹果和3个橘子的例子。

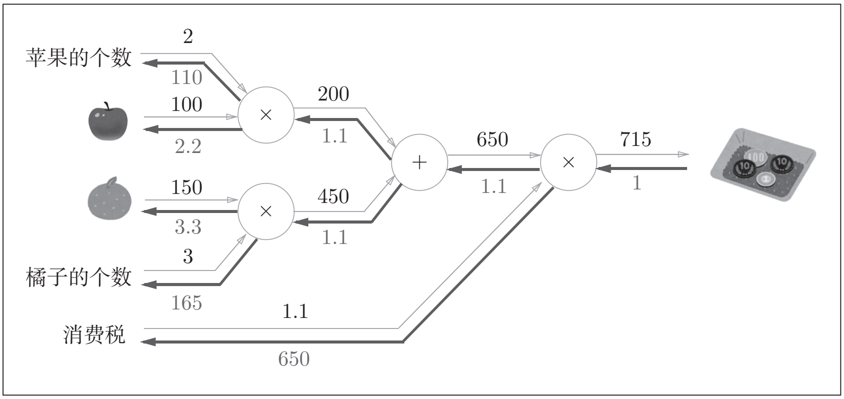


图5-17　购买2个苹果和3个橘子

用Python实现图5-17的计算图的过程如下所示（源代码在ch05/buy_apple_orange.py中）。

```
apple = 100
apple_num = 2
orange = 150
orange_num = 3
tax = 1.1

# layer
mul_apple_layer = MulLayer()
mul_orange_layer = MulLayer()
add_apple_orange_layer = AddLayer()
mul_tax_layer = MulLayer()

# forward
apple_price = mul_apple_layer.forward(apple, apple_num) #(1)
orange_price = mul_orange_layer.forward(orange, orange_num) #(2)
all_price = add_apple_orange_layer.forward(apple_price, orange_price) #(3)
price = mul_tax_layer.forward(all_price, tax) #(4)

# backward
dprice = 1
dall_price, dtax = mul_tax_layer.backward(dprice) #(4)
dapple_price, dorange_price = add_apple_orange_layer.backward(dall_price) #(3)
dorange, dorange_num = mul_orange_layer.backward(dorange_price) #(2)
dapple, dapple_num = mul_apple_layer.backward(dapple_price) #(1)

print(price) # 715
print(dapple_num, dapple, dorange, dorange_num, dtax) # 110 2.2 3.3 165 650
```

这个实现稍微有一点长，但是每一条命令都很简单。首先，生成必要的层，以合适的顺序调用正向传播的`forward()`方法。然后，用与正向传播相反的顺序调用反向传播的`backward()`方法，就可以求出想要的导数。

综上，计算图中层的实现(这里是加法层和乘法层)非常简单，使用这些层可以进行复杂的导数计算。下面，我们来实现神经网络中使用的层。

5.5 激活函数层的实现

现在，我们将计算图的思路应用到神经网络中。这里，我们把构成神经网络的层实现为一个类。先来实现激活函数的`ReLU`层和`Sigmoid`层。

5.5.1 ReLU层

激活函数ReLU(Rectified Linear Unit)由下式(5.7)表示。

$$y = \begin{cases} x & (x > 0) \\ 0 & (x \leqslant 0) \end{cases} \tag{5.7}$$

通过式(5.7)，可以求出y关于x的导数，如式(5.8)所示。

$$\frac{\partial y}{\partial x} = \begin{cases} 1 & (x > 0) \\ 0 & (x \leqslant 0) \end{cases} \tag{5.8}$$

在式(5.8)中，如果正向传播时的输入x大于0，则反向传播会将上游的值原封不动地传给下游。反过来，如果正向传播时的x小于等于0，则反向传播中传给下游的信号将停在此处。用计算图表示的话，如图5-18所示。

现在我们来实现`ReLU`层。在神经网络的层的实现中，一般假定`forward()`和`backward()`的参数是NumPy数组。另外，实现`ReLU`层的源代码在`common/layers.py`中。

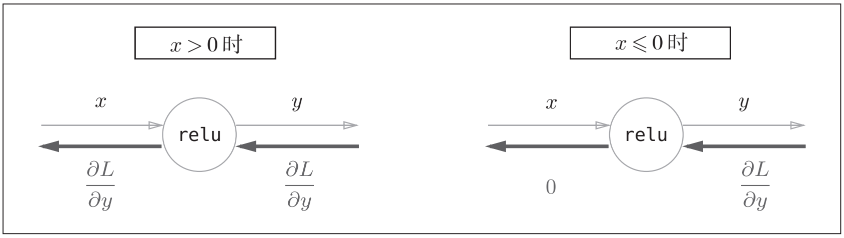


图5-18 ReLU层的计算图

```
class Relu:
    def __init__(self):
        self.mask = None

    def forward(self, x):
        self.mask = (x <= 0)
        out = x.copy()
        out[self.mask] = 0

        return out

    def backward(self, dout):
        dout[self.mask] = 0
        dx = dout

        return dx
```

Relu类有实例变量mask。这个变量mask是由True/False构成的NumPy数组，它会把正向传播时的输入x的元素中小于等于0的地方保存为True，其他地方(大于0的元素)保存为False。如下例所示，mask变量保存了由True/False构成的NumPy数组。

```
>>> x = np.array( [[1.0, -0.5], [-2.0, 3.0]] )
>>> print(x)
[[ 1.  -0.5]
 [-2.   3. ]]
>>> mask = (x <= 0)
>>> print(mask)
[[False  True]
 [ True False]]
```

如图5-18所示，如果正向传播时的输入值小于等于0，则反向传播的值为0。因此，反向传播中会使用正向传播时保存的mask，将从上游传来的dout的mask中的元素为True的地方设为0。



ReLU层的作用就像电路中的开关一样。正向传播时，有电流通过的话，就将开关设为ON；没有电流通过的话，就将开关设为OFF。反向传播时，开关为ON的话，电流会直接通过；开关为OFF的话，则不会有电流通过。

5.5.2 Sigmoid层

接下来，我们来实现sigmoid函数。sigmoid函数由式(5.9)表示。

$$y = \frac{1}{1 + \exp(-x)} \tag{5.9}$$

用计算图表示式(5.9)的话，则如图5-19所示。

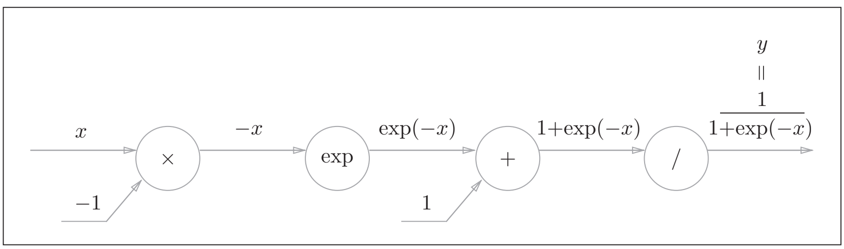


图5-19 sigmoid层的计算图（仅正向传播）

图5-19中，除了“×”和“+”节点外，还出现了新的“exp”和“/”节点。“exp”节点会进行$y = \exp(x)$的计算，“/”节点会进行$y = \frac{1}{x}$的计算。

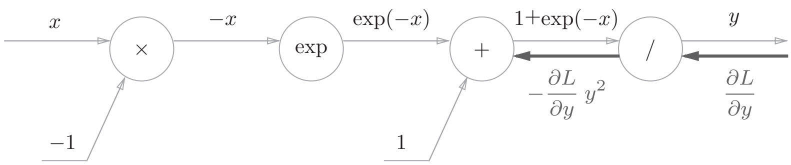
如图5-19所示，式(5.9)的计算由局部计算的传播构成。下面我们就来进行图5-19的计算图的反向传播。这里，作为总结，我们来依次看一下反向传播的流程。

步骤1

“/” 节点表示 $y=\frac{1}{x}$，它的导数可以解析性地表示为下式。

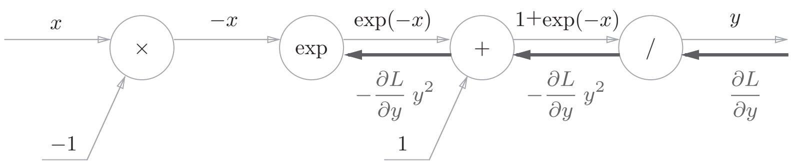
$$\begin{aligned}\frac{\partial y}{\partial x} &= -\frac{1}{x^2} \\ &= -y^2\end{aligned} \tag{5.10}$$

根据式(5.10)，反向传播时，会将上游的值乘以 $-y^2$(正向传播的输出的平方乘以 -1 后的值)后，再传给下游。计算图如下所示。



步骤2

“+” 节点将上游的值原封不动地传给下游。计算图如下所示。

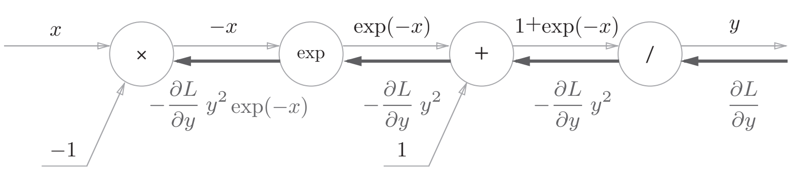


步骤3

“exp” 节点表示 $y=\exp(x)$，它的导数由下式表示。

$$\frac{\partial y}{\partial x} = \exp(x) \tag{5.11}$$

计算图中，上游的值乘以正向传播时的输出(这个例子中是 $\exp(-x)$)后，再传给下游。



步骤4

“×”节点将正向传播时的值翻转后做乘法运算。因此，这里要乘以 -1。

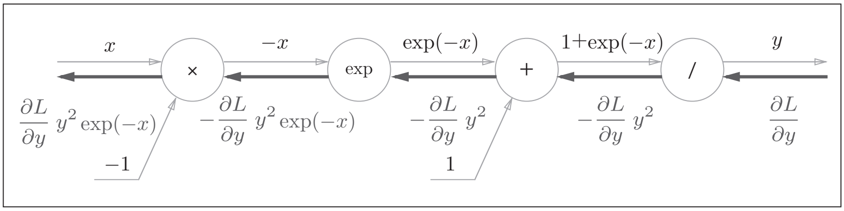


图5-20 Sigmoid层的计算图

根据上述内容，图5-20的计算图可以进行Sigmoid层的反向传播。从图5-20的结果可知，反向传播的输出为 $\frac{\partial L}{\partial y}y^2\exp(-x)$，这个值会传播给下游的节点。这里要注意，$\frac{\partial L}{\partial y}y^2\exp(-x)$ 这个值只根据正向传播时的输入 x 和输出 y 就可以算出来。因此，图5-20的计算图可以画成图5-21的集约化的“sigmoid”节点。

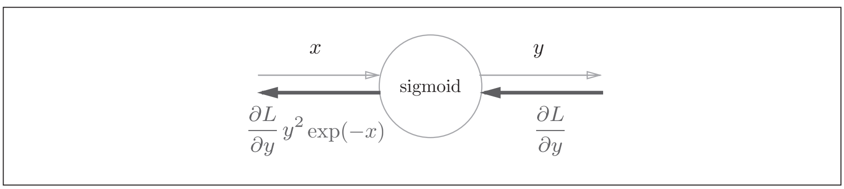


图5-21 Sigmoid层的计算图（简洁版）

图5-20的计算图和简洁版的图5-21的计算图的计算结果是相同的，但是，简洁版的计算图可以省略反向传播中的计算过程，因此计算效率更高。此外，通过对节点进行集约化，可以不用在意Sigmoid层中琐碎的细节，而只需要专注它的输入和输出，这一点也很重要。

另外，$\frac{\partial L}{\partial y}y^2\exp(-x)$ 可以进一步整理如下。

$$
\begin{aligned}
\frac{\partial L}{\partial y}y^2\exp(-x) &= \frac{\partial L}{\partial y}\frac{1}{(1+\exp(-x))^2}\exp(-x) \\
&= \frac{\partial L}{\partial y}\frac{1}{1+\exp(-x)}\frac{\exp(-x)}{1+\exp(-x)} \\
&= \frac{\partial L}{\partial y}y(1-y)
\end{aligned}
\tag{5.12}
$$

因此，图5-21所表示的Sigmoid层的反向传播，只根据正向传播的输出就能计算出来。

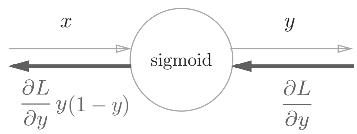


图5-22 Sigmoid层的计算图：可以根据正向传播的输出 y 计算反向传播

现在，我们用Python实现Sigmoid层。参考图5-22，可以像下面这样实现（实现的代码在`common/layers.py`中）。

```
class Sigmoid:
    def __init__(self):
        self.out = None

    def forward(self, x):
        out = 1 / (1 + np.exp(-x))
        self.out = out

        return out

    def backward(self, dout):
        dx = dout * (1.0 - self.out) * self.out

        return dx
```

这个实现中，正向传播时将输出保存在了实例变量`out`中。然后，反向传播时，使用该变量`out`进行计算。

5.6 Affine/Softmax层的实现

5.6.1 Affine层

神经网络的正向传播中，为了计算加权信号的总和，使用了矩阵的乘积运算（NumPy中是`np.dot()`，具体请参照3.3节）。比如，还记得我们用Python进行了下面的实现吗？

```
>>> X = np.random.rand(2)   # 输入
>>> W = np.random.rand(2,3) # 权重
>>> B = np.random.rand(3)   # 偏置
>>>
>>> X.shape # (2,)
>>> W.shape # (2, 3)
>>> B.shape # (3,)
>>>
>>> Y = np.dot(X, W) + B
```

这里，$\boldsymbol{X}$、$\boldsymbol{W}$、$\boldsymbol{B}$ 分别是形状为 (2,)、(2, 3)、(3,) 的多维数组。这样一来，神经元的加权和可以用`Y = np.dot(X, W) + B`计算出来。然后，$\boldsymbol{Y}$ 经过激活函数转换后，传递给下一层。这就是神经网络正向传播的流程。此外，我们来复习一下，矩阵的乘积运算的要点是使对应维度的元素个数一致。比如，如下面的图 5-23 所示，$\boldsymbol{X}$ 和 $\boldsymbol{W}$ 的乘积必须使对应维度的元素个数一致。另外，这里矩阵的形状用 (2, 3) 这样的括号表示（为了和 NumPy 的 `shape` 属性的输出一致）。

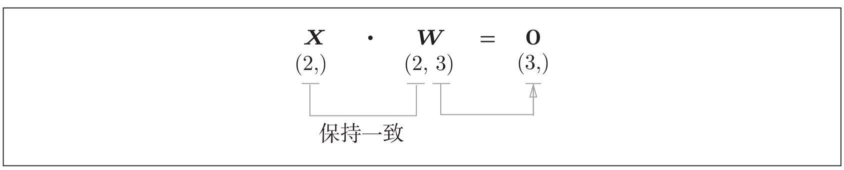


图 5-23　矩阵的乘积运算中对应维度的元素个数要保持一致

神经网络的正向传播中进行的矩阵的乘积运算在几何学领域被称为“仿射变换”[①]。因此，这里将进行仿射变换的处理实现为“Affine层”。

现在将这里进行的求矩阵的乘积与偏置的和的运算用计算图表示出来。将乘积运算用“dot”节点表示的话，则`np.dot(X, W) + B`的运算可用图 5-24 所示的计算图表示出来。另外，在各个变量的上方标记了它们的形状（比如，计算图上显示了 $\boldsymbol{X}$ 的形状为 (2,)，$\boldsymbol{X} \cdot \boldsymbol{W}$ 的形状为 (3,) 等）。

① 几何中，仿射变换包括一次线性变换和一次平移，分别对应神经网络的加权和运算与加偏置运算。——译者注

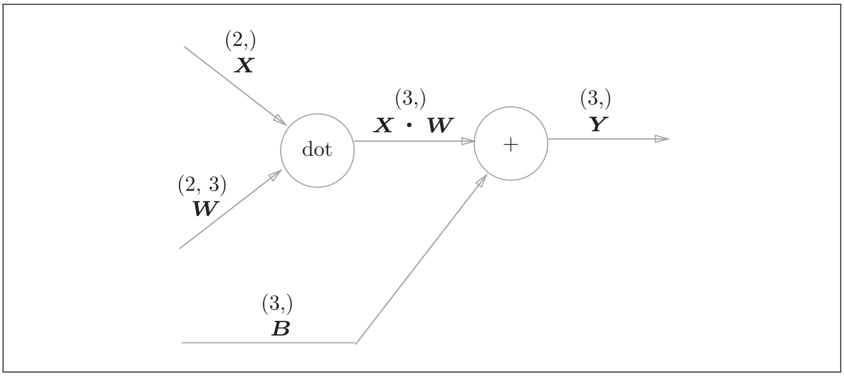


图5-24　Affine层的计算图(注意变量是矩阵，各个变量的上方标记了该变量的形状)

图5-24是比较简单的计算图，不过要注意 $\boldsymbol{X}$、$\boldsymbol{W}$、$\boldsymbol{B}$ 是矩阵(多维数组)。之前我们见到的计算图中各个节点间流动的是标量，而这个例子中各个节点间传播的是矩阵。

现在我们来考虑图5-24的计算图的反向传播。以矩阵为对象的反向传播，按矩阵的各个元素进行计算时，步骤和以标量为对象的计算图相同。实际写一下的话，可以得到下式(这里省略了式(5.13)的推导过程)。

$$
\begin{aligned}
\frac{\partial L}{\partial \boldsymbol{X}} &= \frac{\partial L}{\partial \boldsymbol{Y}} \cdot \boldsymbol{W}^{\mathrm{T}} \\
\frac{\partial L}{\partial \boldsymbol{W}} &= \boldsymbol{X}^{\mathrm{T}} \cdot \frac{\partial L}{\partial \boldsymbol{Y}}
\end{aligned}
\tag{5.13}
$$

式(5.13)中 $\boldsymbol{W}^{\mathrm{T}}$ 的T表示转置。转置操作会把 $\boldsymbol{W}$ 的元素 (i, j) 换成元素 (j, i)。用数学式表示的话，可以写成下面这样。

$$
\begin{aligned}
\boldsymbol{W}^{\mathrm{T}} &= \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{pmatrix} \\
\boldsymbol{W}^{\mathrm{T}} &= \begin{pmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \\ w_{13} & w_{23} \end{pmatrix}
\end{aligned}
\tag{5.14}
$$

如式(5.14)所示，如果 $\boldsymbol{W}$ 的形状是(2, 3)，$\boldsymbol{W}^{\mathrm{T}}$ 的形状就是(3, 2)。

现在，我们根据式(5.13)，尝试写出计算图的反向传播，如图5-25所示。

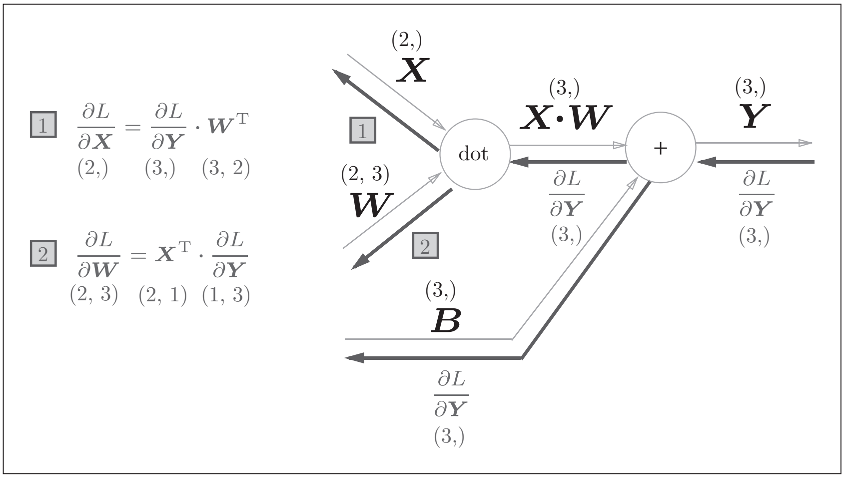


图5-25 Affine层的反向传播：注意变量是多维数组。反向传播时各个变量的下方标记了该变量的形状

我们看一下图2-25的计算图中各个变量的形状。尤其要注意，$\boldsymbol{X}$ 和 $\frac{\partial L}{\partial \boldsymbol{X}}$ 形状相同，$\boldsymbol{W}$ 和 $\frac{\partial L}{\partial \boldsymbol{W}}$ 形状相同。从下面的数学式可以很明确地看出 $\boldsymbol{X}$ 和 $\frac{\partial L}{\partial \boldsymbol{X}}$ 形状相同。

$$
\begin{aligned}
\boldsymbol{X} &= (x_0, x_1, \cdots, x_n) \\
\frac{\partial L}{\partial \boldsymbol{X}} &= \left(\frac{\partial L}{\partial x_0}, \frac{\partial L}{\partial x_1}, \cdots, \frac{\partial L}{\partial x_n} \right)
\end{aligned} \tag{5.15}
$$

为什么要注意矩阵的形状呢？因为矩阵的乘积运算要求对应维度的元素个数保持一致，通过确认一致性，就可以导出式(5.13)。比如，$\frac{\partial L}{\partial \boldsymbol{Y}}$ 的形状是(3,)，$\boldsymbol{W}$ 的形状是(2, 3)时，思考 $\frac{\partial L}{\partial \boldsymbol{Y}}$ 和 $\boldsymbol{W}^{\mathrm{T}}$ 的乘积，使得 $\frac{\partial L}{\partial \boldsymbol{X}}$ 的形状为(2,)(图5-26)。这样一来，就会自然而然地推导出式(5.13)。

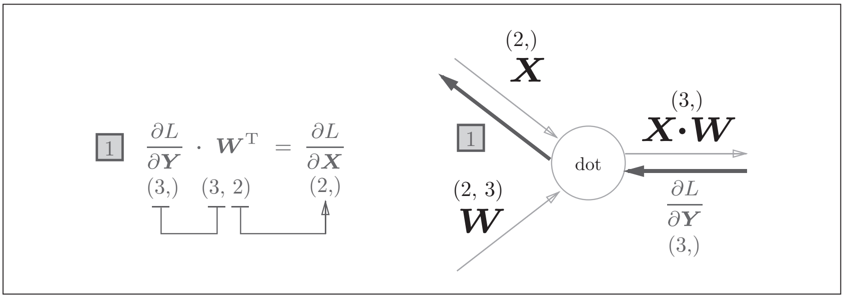


图5-26 矩阵的乘积（“dot”节点）的反向传播可以通过组建使矩阵对应维度的元素个数一致的乘积运算而推导出来

5.6.2 批版本的Affine层

前面介绍的Affine层的输入 $\boldsymbol{X}$ 是以单个数据为对象的。现在我们考虑 N 个数据一起进行正向传播的情况，也就是批版本的Affine层。

先给出批版本的Affine层的计算图，如图5-27所示。

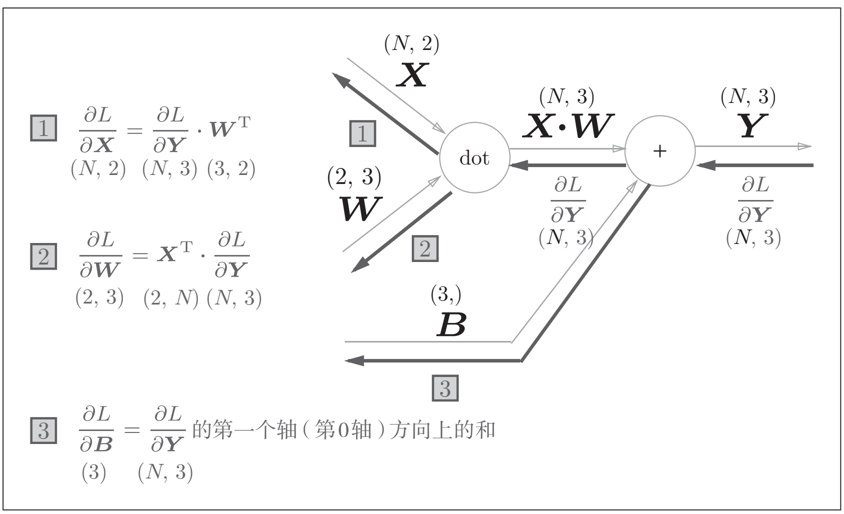


图5-27 批版本的Affine层的计算图

与刚刚不同的是，现在输入 $\boldsymbol{X}$ 的形状是 $(N, 2)$。之后就和前面一样，在计算图上进行单纯的矩阵计算。反向传播时，如果注意矩阵的形状，就可以和前面一样推导出 $\frac{\partial L}{\partial \boldsymbol{X}}$ 和 $\frac{\partial L}{\partial \boldsymbol{W}}$。

加上偏置时，需要特别注意。正向传播时，偏置被加到 $\boldsymbol{X} \cdot \boldsymbol{W}$ 的各个数据上。比如，$N = 2$（数据为2个）时，偏置会被分别加到这2个数据（各自的计算结果）上，具体的例子如下所示。

```
>>> X_dot_W = np.array([[0, 0, 0], [10, 10, 10]])
>>> B = np.array([1, 2, 3])
>>>
>>> X_dot_W
array([[ 0,  0,  0],
       [ 10, 10, 10]])
>>> X_dot_W + B
array([[ 1,  2,  3],
       [11, 12, 13]])
```

正向传播时，偏置会被加到每一个数据（第1个、第2个……）上。因此，反向传播时，各个数据的反向传播的值需要汇总为偏置的元素。用代码表示的话，如下所示。

```
>>> dY = np.array([[1, 2, 3,], [4, 5, 6]])
>>> dY
array([[1, 2, 3],
       [4, 5, 6]])
>>>
>>> dB = np.sum(dY, axis=0)
>>> dB
array([5, 7, 9])
```

这个例子中，假定数据有2个（$N = 2$）。偏置的反向传播会对这2个数据的导数按元素进行求和。因此，这里使用了 `np.sum()` 对第0轴（以数据为单位的轴，`axis=0`）方向上的元素进行求和。

综上所述，Affine的实现如下所示。另外，`common/layers.py` 中的Affine的实现考虑了输入数据为张量（四维数据）的情况，与这里介绍的稍有差别。

```
class Affine:
    def __init__(self, W, b):
        self.W = W
        self.b = b
        self.x = None
        self.dW = None
        self.db = None

    def forward(self, x):
        self.x = x
        out = np.dot(x, self.W) + self.b

        return out

    def backward(self, dout):
        dx = np.dot(dout, self.W.T)
        self.dW = np.dot(self.x.T, dout)
        self.db = np.sum(dout, axis=0)

        return dx
```

5.6.3　Softmax-with-Loss 层

最后介绍一下输出层的softmax函数。前面我们提到过，softmax函数会将输入值正规化之后再输出。比如手写数字识别时，Softmax层的输出如图5-28所示。

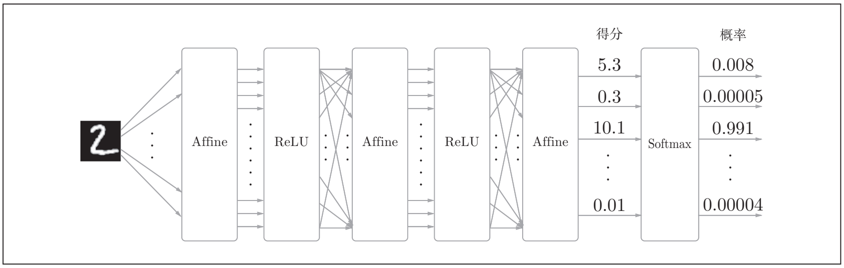


图5-28　输入图像通过Affine层和ReLU层进行转换，10个输入通过Softmax层进行正规化。在这个例子中，"0"的得分是5.3，这个值经过Softmax层转换为0.008（0.8%）；"2"的得分是10.1，被转换为0.991（99.1%）

在图5-28中，Softmax层将输入值正规化（将输出值的和调整为1）之后再输出。另外，因为手写数字识别要进行10类分类，所以向Softmax层的输入也有10个。

神经网络中进行的处理有**推理**（inference）和**学习**两个阶段。神经网络的推理通常不使用Softmax层。比如，用图5-28的网络进行推理时，会将最后一个Affine层的输出作为识别结果。神经网络中未被正规化的输出结果（图5-28中Softmax层前面的Affine层的输出）有时被称为"得分"。也就是说，当神经网络的推理只需要给出一个答案的情况下，因为此时只对得分最大值感兴趣，所以不需要Softmax层。不过，神经网络的学习阶段则需要Softmax层。

下面来实现Softmax层。考虑到这里也包含作为损失函数的交叉熵误差（cross entropy error），所以称为"Softmax-with-Loss层"。Softmax-with-Loss层（Softmax函数和交叉熵误差）的计算图如图5-29所示。

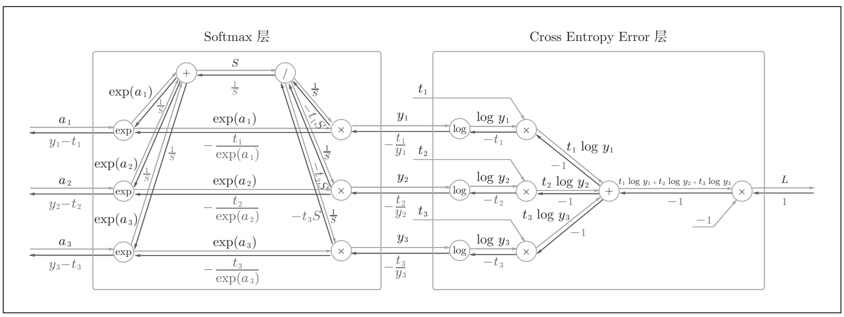


图5-29 Softmax-with-Loss层的计算图

可以看到，Softmax-with-Loss层有些复杂。这里只给出了最终结果，对Softmax-with-Loss层的导出过程感兴趣的读者，请参照附录A。

图5-29的计算图可以简化成图5-30。

图5-30的计算图中，softmax函数记为Softmax层，交叉熵误差记为Cross Entropy Error层。这里假设要进行3类分类，从前面的层接收3个输入（得分）。如图5-30所示，Softmax层将输入（a_1, a_2, a_3）正规化，输出（y_1, y_2, y_3）。Cross Entropy Error层接收Softmax的输出（y_1, y_2, y_3）和教师标签（t_1, t_2, t_3），从这些数据中输出损失L。

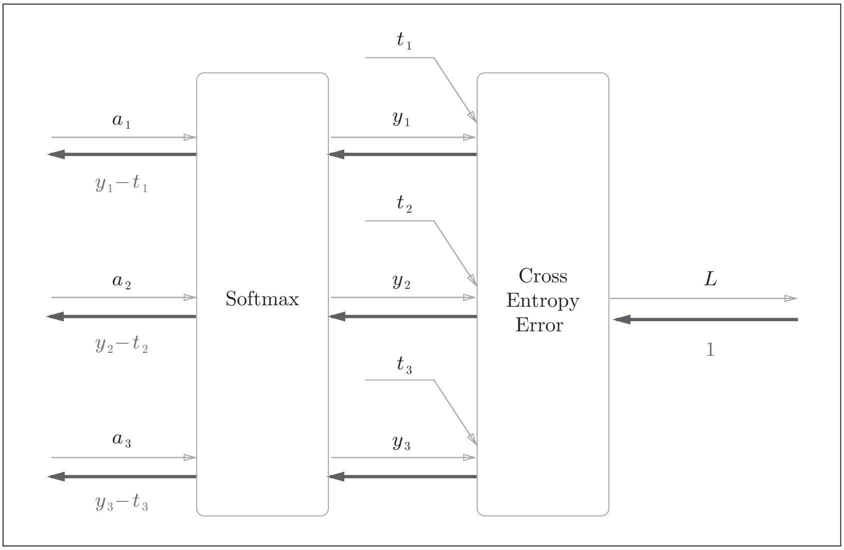


图5-30 “简易版”的Softmax-with-Loss层的计算图

图5-30中要注意的是反向传播的结果。Softmax层的反向传播得到了$(y_1-t_1, y_2-t_2, y_3-t_3)$这样“漂亮”的结果。由于$(y_1, y_2, y_3)$是Softmax层的输出，$(t_1, t_2, t_3)$是监督数据，所以$(y_1-t_1, y_2-t_2, y_3-t_3)$是Softmax层的输出和教师标签的差分。神经网络的反向传播会把这个差分表示的误差传递给前面的层，这是神经网络学习中的重要性质。

神经网络学习的目的就是通过调整权重参数，使神经网络的输出（Softmax的输出）接近教师标签。因此，必须将神经网络的输出与教师标签的误差高效地传递给前面的层。刚刚的$(y_1-t_1, y_2-t_2, y_3-t_3)$正是Softmax层的输出与教师标签的差，直截了当地表示了当前神经网络的输出与教师标签的误差。

这里考虑一个具体的例子，比如思考教师标签是$(0, 1, 0)$，Softmax层的输出是$(0.3, 0.2, 0.5)$的情形。因为正确解标签处的概率是0.2（20%），这个时候的神经网络未能进行正确的识别。此时，Softmax层的反向传播传递的是$(0.3, -0.8, 0.5)$这样一个大的误差。因为这个大的误差会向前面的层传播，所以Softmax层前面的层会从这个大的误差中学习到“大”的内容。

使用交叉熵误差作为softmax函数的损失函数后，反向传播得到 $(y_1 - t_1, y_2 - t_2, y_3 - t_3)$ 这样“漂亮”的结果。实际上，这样“漂亮”的结果并不是偶然的，而是为了得到这样的结果，特意设计了交叉熵误差函数。回归问题中输出层使用“恒等函数”，损失函数使用“平方和误差”，也是出于同样的理由（3.5节）。也就是说，使用“平方和误差”作为“恒等函数”的损失函数，反向传播才能得到 $(y_1 - t_1, y_2 - t_2, y_3 - t_3)$ 这样“漂亮”的结果。

再举一个例子，比如思考教师标签是 $(0, 1, 0)$，Softmax层的输出是 $(0.01, 0.99, 0)$ 的情形（这个神经网络识别得相当准确）。此时Softmax层的反向传播传递的是 $(0.01, -0.01, 0)$ 这样一个小的误差。这个小的误差也会向前面的层传播，因为误差很小，所以Softmax层前面的层学到的内容也很“小”。

现在来进行Softmax-with-Loss层的实现，实现过程如下所示。

```
class SoftmaxWithLoss:
    def __init__(self):
        self.loss = None # 损失
        self.y = None    # softmax的输出
        self.t = None    # 监督数据（one-hot vector）

    def forward(self, x, t):
        self.t = t
        self.y = softmax(x)
        self.loss = cross_entropy_error(self.y, self.t)

        return self.loss

    def backward(self, dout=1):
        batch_size = self.t.shape[0]
        dx = (self.y - self.t) / batch_size

        return dx
```

这个实现利用了3.5.2节和4.2.4节中实现的`softmax()`和`cross_entropy_error()`函数。因此，这里的实现非常简单。请注意反向传播时，将要传播的值除以批的大小（`batch_size`）后，传递给前面的层的是单个数据的误差。

5.7 误差反向传播法的实现

通过像组装乐高积木一样组装上一节中实现的层，可以构建神经网络。本节我们将通过组装已经实现的层来构建神经网络。

5.7.1 神经网络学习的全貌图

在进行具体的实现之前，我们再来确认一下神经网络学习的全貌图。神经网络学习的步骤如下所示。

前提

神经网络中有合适的权重和偏置，调整权重和偏置以便拟合训练数据的过程称为学习。神经网络的学习分为下面4个步骤。

步骤1（mini-batch）

从训练数据中随机选择一部分数据。

步骤2（计算梯度）

计算损失函数关于各个权重参数的梯度。

步骤3（更新参数）

将权重参数沿梯度方向进行微小的更新。

步骤4（重复）

重复步骤1、步骤2、步骤3。

之前介绍的误差反向传播法会在步骤2中出现。上一章中，我们利用数值微分求得了这个梯度。数值微分虽然实现简单，但是计算要耗费较多的时间。和需要花费较多时间的数值微分不同，误差反向传播法可以快速高效地计算梯度。

5.7.2 对应误差反向传播法的神经网络的实现

现在来进行神经网络的实现。这里我们要把2层神经网络实现为TwoLayerNet。首先，将这个类的实例变量和方法整理成表5-1和表5-2。

表5-1 TwoLayerNet类的实例变量

实例变量	说明
params	保存神经网络的参数的字典型变量。 params['W1']是第1层的权重，params['b1']是第1层的偏置。 params['W2']是第2层的权重，params['b2']是第2层的偏置
layers	保存神经网络的层的**有序字典型变量**。 以layers['Affine1']、layers['ReLu1']、layers['Affine2']的形式，通过有序字典保存各个层
lastLayer	神经网络的最后一层。 本例中为SoftmaxWithLoss层

表5-2 TwoLayerNet类的方法

方法	说明
__init__(self, input_size, hidden_size, output_size, weight_init_std)	进行初始化。 参数从头开始依次是输入层的神经元数、隐藏层的神经元数、输出层的神经元数、初始化权重时的高斯分布的规模
predict(self, x)	进行识别（推理）。 参数x是图像数据
loss(self, x, t)	计算损失函数的值。 参数x是图像数据、t是正确解标签
accuracy(self, x, t)	计算识别精度
numerical_gradient(self, x, t)	通过数值微分计算关于权重参数的梯度（同上一章）
gradient(self, x, t)	通过误差反向传播法计算关于权重参数的梯度

这个类的实现稍微有一点长，但是内容和4.5节的学习算法的实现有很多共通的部分，不同点主要在于这里使用了层。通过使用层，获得识别结果的处理（predict()）和计算梯度的处理（gradient()）只需通过层之间的传递就

能完成。下面是TwoLayerNet的代码实现。

```
import sys, os
sys.path.append(os.pardir)
import numpy as np
from common.layers import *
from common.gradient import numerical_gradient
from collections import OrderedDict

class TwoLayerNet:

    def __init__(self, input_size, hidden_size, output_size,
                 weight_init_std=0.01):
        # 初始化权重
        self.params = {}
        self.params['W1'] = weight_init_std * \
                            np.random.randn(input_size, hidden_size)
        self.params['b1'] = np.zeros(hidden_size)
        self.params['W2'] = weight_init_std * \
                            np.random.randn(hidden_size, output_size)
        self.params['b2'] = np.zeros(output_size)

        # 生成层
        self.layers = OrderedDict()
        self.layers['Affine1'] = \
            Affine(self.params['W1'], self.params['b1'])
        self.layers['Relu1'] = Relu()
        self.layers['Affine2'] = \
            Affine(self.params['W2'], self.params['b2'])

        self.lastLayer = SoftmaxWithLoss()

    def predict(self, x):
        for layer in self.layers.values():
            x = layer.forward(x)

        return x

    # x:输入数据, t:监督数据
    def loss(self, x, t):
        y = self.predict(x)
        return self.lastLayer.forward(y, t)

    def accuracy(self, x, t):
        y = self.predict(x)
        y = np.argmax(y, axis=1)
        if t.ndim != 1 : t = np.argmax(t, axis=1)
```

```
        accuracy = np.sum(y == t) / float(x.shape[0])
        return accuracy

    # x:输入数据, t:监督数据
    def numerical_gradient(self, x, t):
        loss_W = lambda W: self.loss(x, t)

        grads = {}
        grads['W1'] = numerical_gradient(loss_W, self.params['W1'])
        grads['b1'] = numerical_gradient(loss_W, self.params['b1'])
        grads['W2'] = numerical_gradient(loss_W, self.params['W2'])
        grads['b2'] = numerical_gradient(loss_W, self.params['b2'])

        return grads

    def gradient(self, x, t):
        # forward
        self.loss(x, t)

        # backward
        dout = 1
        dout = self.lastLayer.backward(dout)

        layers = list(self.layers.values())
        layers.reverse()
        for layer in layers:
            dout = layer.backward(dout)

        # 设定
        grads = {}
        grads['W1'] = self.layers['Affine1'].dW
        grads['b1'] = self.layers['Affine1'].db
        grads['W2'] = self.layers['Affine2'].dW
        grads['b2'] = self.layers['Affine2'].db

        return grads
```

请注意这个实现中的粗体字代码部分，尤其是将神经网络的层保存为`OrderedDict`这一点非常重要。`OrderedDict`是有序字典，“有序”是指它可以记住向字典里添加元素的顺序。因此，神经网络的正向传播只需按照添加元素的顺序调用各层的`forward()`方法就可以完成处理，而反向传播只需要按照相反的顺序调用各层即可。因为Affine层和ReLU层的内部会正确处理正向传播和反向传播，所以这里要做的事情仅仅是以正确的顺序连接各层，再按顺序（或者逆序）调用各层。

像这样通过将神经网络的组成元素以层的方式实现，可以轻松地构建神经网络。这个用层进行模块化的实现具有很大优点。因为想另外构建一个神经网络（比如5层、10层、20层……的大的神经网络）时，只需像组装乐高积木那样添加必要的层就可以了。之后，通过各个层内部实现的正向传播和反向传播，就可以正确计算进行识别处理或学习所需的梯度。

5.7.3 误差反向传播法的梯度确认

到目前为止，我们介绍了两种求梯度的方法。一种是基于数值微分的方法，另一种是解析性地求解数学式的方法。后一种方法通过使用误差反向传播法，即使存在大量的参数，也可以高效地计算梯度。因此，后文将不再使用耗费时间的数值微分，而是使用误差反向传播法求梯度。

数值微分的计算很耗费时间，而且如果有误差反向传播法的（正确的）实现的话，就没有必要使用数值微分的实现了。那么数值微分有什么用呢？实际上，在确认误差反向传播法的实现是否正确时，是需要用到数值微分的。

数值微分的优点是实现简单，因此，一般情况下不太容易出错。而误差反向传播法的实现很复杂，容易出错。所以，经常会比较数值微分的结果和误差反向传播法的结果，以确认误差反向传播法的实现是否正确。确认数值微分求出的梯度结果和误差反向传播法求出的结果是否一致（严格地讲，是非常相近）的操作称为**梯度确认**（gradient check）。梯度确认的代码实现如下所示（源代码在`ch05/gradient_check.py`中）。

```
import sys, os
sys.path.append(os.pardir)
import numpy as np
from dataset.mnist import load_mnist
from two_layer_net import TwoLayerNet

# 读入数据
(x_train, t_train), (x_test, t_test) = \ load_mnist(normalize=True, one_
hot_label = True)

network = TwoLayerNet(input_size=784, hidden_size=50, output_size=10)

x_batch = x_train[:3]
```

```
t_batch = t_train[:3]

grad_numerical = network.numerical_gradient(x_batch, t_batch)
grad_backprop = network.gradient(x_batch, t_batch)

# 求各个权重的绝对误差的平均值
for key in grad_numerical.keys():
    diff = np.average( np.abs(grad_backprop[key] - grad_numerical[key]) )
    print(key + ":" + str(diff))
```

和以前一样，读入MNIST数据集。然后，使用训练数据的一部分，确认数值微分求出的梯度和误差反向传播法求出的梯度的误差。这里误差的计算方法是求各个权重参数中对应元素的差的绝对值，并计算其平均值。运行上面的代码后，会输出如下结果。

```
b1:9.70418809871e-13
W2:8.41139039497e-13
b2:1.1945999745e-10
W1:2.2232446644e-13
```

从这个结果可以看出，通过数值微分和误差反向传播法求出的梯度的差非常小。比如，第1层的偏置的误差是9.7e-13（0.00000000000097）。这样一来，我们就知道了通过误差反向传播法求出的梯度是正确的，误差反向传播法的实现没有错误。

数值微分和误差反向传播法的计算结果之间的误差为0是很少见的。这是因为计算机的计算精度有限（比如，32位浮点数）。受到数值精度的限制，刚才的误差一般不会为0，但是如果实现正确的话，可以期待这个误差是一个接近0的很小的值。如果这个值很大，就说明误差反向传播法的实现存在错误。

5.7.4　使用误差反向传播法的学习

最后，我们来看一下使用了误差反向传播法的神经网络的学习的实现。和之前的实现相比，不同之处仅在于通过误差反向传播法求梯度这一点。这里只列出了代码，省略了说明（源代码在ch05/train_neuralnet.py中）。

```
import sys, os
sys.path.append(os.pardir)
import numpy as np
from dataset.mnist import load_mnist
from two_layer_net import TwoLayerNet

# 读入数据
(x_train, t_train), (x_test, t_test) = \
    load_mnist(normalize=True, one_hot_label=True)

network = TwoLayerNet(input_size=784, hidden_size=50, output_size=10)

iters_num = 10000
train_size = x_train.shape[0]
batch_size = 100
learning_rate = 0.1
train_loss_list = []
train_acc_list = []
test_acc_list = []

iter_per_epoch = max(train_size / batch_size, 1)

for i in range(iters_num):
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    t_batch = t_train[batch_mask]

    # 通过误差反向传播法求梯度
    grad = network.gradient(x_batch, t_batch)

    # 更新
    for key in ('W1', 'b1', 'W2', 'b2'):
        network.params[key] -= learning_rate * grad[key]

    loss = network.loss(x_batch, t_batch)
    train_loss_list.append(loss)

    if i % iter_per_epoch == 0:
        train_acc = network.accuracy(x_train, t_train)
        test_acc = network.accuracy(x_test, t_test)
        train_acc_list.append(train_acc)
        test_acc_list.append(test_acc)
        print(train_acc, test_acc)
```

5.8 小结

本章我们介绍了将计算过程可视化的计算图，并使用计算图，介绍了神经网络中的误差反向传播法，并以层为单位实现了神经网络中的处理。我们学过的层有ReLU层、Softmax-with-Loss层、Affine层、Softmax层等，这些层中实现了`forward`和`backward`方法，通过将数据正向和反向地传播，可以高效地计算权重参数的梯度。通过使用层进行模块化，神经网络中可以自由地组装层，轻松构建出自己喜欢的网络。

本章所学的内容

- 通过使用计算图，可以直观地把握计算过程。
- 计算图的节点是由局部计算构成的。局部计算构成全局计算。
- 计算图的正向传播进行一般的计算。通过计算图的反向传播，可以计算各个节点的导数。
- 通过将神经网络的组成元素实现为层，可以高效地计算梯度（反向传播法）。
- 通过比较数值微分和误差反向传播法的结果，可以确认误差反向传播法的实现是否正确（梯度确认）。

第6章 与学习相关的技巧

本章将介绍神经网络的学习中的一些重要观点，主题涉及寻找最优权重参数的最优化方法、权重参数的初始值、超参数的设定方法等。此外，为了应对过拟合，本章还将介绍权值衰减、Dropout等正则化方法，并进行实现。最后将对近年来众多研究中使用的Batch Normalization方法进行简单的介绍。使用本章介绍的方法，可以高效地进行神经网络（深度学习）的学习，提高识别精度。让我们一起往下看吧！

6.1 参数的更新

神经网络的学习的目的是找到使损失函数的值尽可能小的参数。这是寻找最优参数的问题，解决这个问题的过程称为**最优化**（optimization）。遗憾的是，神经网络的最优化问题非常难。这是因为参数空间非常复杂，无法轻易找到最优解（无法使用那种通过解数学式一下子就求得最小值的方法）。而且，在深度神经网络中，参数的数量非常庞大，导致最优化问题更加复杂。

在前几章中，为了找到最优参数，我们将参数的梯度（导数）作为了线索。使用参数的梯度，沿梯度方向更新参数，并重复这个步骤多次，从而逐渐靠近最优参数，这个过程称为**随机梯度下降法**（stochastic gradient descent），简称**SGD**。SGD是一个简单的方法，不过比起胡乱地搜索参数空间，也算是“聪明”的方法。但是，根据不同的问题，也存在比SGD更加聪明的方法。本节

我们将指出SGD的缺点，并介绍SGD以外的其他最优化方法。

6.1.1 探险家的故事

进入正题前，我们先打一个比方，来说明关于最优化我们所处的状况。

> 有一个性情古怪的探险家。他在广袤的干旱地带旅行，坚持寻找幽深的山谷。他的目标是要到达最深的谷底(他称之为“至深之地”)。这也是他旅行的目的。并且，他给自己制定了两个严格的“规定”：一个是不看地图；另一个是把眼睛蒙上。因此，他并不知道最深的谷底在这个广袤的大地的何处，而且什么也看不见。在这么严苛的条件下，这位探险家如何前往“至深之地”呢？他要如何迈步，才能迅速找到“至深之地”呢？

寻找最优参数时，我们所处的状况和这位探险家一样，是一个漆黑的世界。我们必须在没有地图、不能睁眼的情况下，在广袤、复杂的地形中寻找“至深之地”。大家可以想象这是一个多么难的问题。

在这么困难的状况下，地面的坡度显得尤为重要。探险家虽然看不到周围的情况，但是能够知道当前所在位置的坡度（通过脚底感受地面的倾斜状况）。于是，朝着当前所在位置的坡度最大的方向前进，就是SGD的策略。勇敢的探险家心里可能想着只要重复这一策略，总有一天可以到达“至深之地”。

6.1.2 SGD

让大家感受了最优化问题的难度之后，我们再来复习一下SGD。用数学式可以将SGD写成如下的式(6.1)。

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial L}{\partial \boldsymbol{W}} \tag{6.1}$$

这里把需要更新的权重参数记为$\boldsymbol{W}$，把损失函数关于$\boldsymbol{W}$的梯度记为$\frac{\partial L}{\partial \boldsymbol{W}}$。$\eta$表示学习率，实际上会取0.01或0.001这些事先决定好的值。式子中的$\leftarrow$

表示用右边的值更新左边的值。如式(6.1)所示，SGD是朝着梯度方向只前进一定距离的简单方法。现在，我们将SGD实现为一个Python类(为方便后面使用，我们将其实现为一个名为`SGD`的类)。

```
class SGD:
    def __init__(self, lr=0.01):
        self.lr = lr

    def update(self, params, grads):
        for key in params.keys():
            params[key] -= self.lr * grads[key]
```

这里，进行初始化时的参数`lr`表示learning rate(学习率)。这个学习率会保存为实例变量。此外，代码段中还定义了`update(params, grads)`方法，这个方法在SGD中会被反复调用。参数`params`和`grads`(与之前的神经网络的实现一样)是字典型变量，按`params['W1']`、`grads['W1']`的形式，分别保存了权重参数和它们的梯度。

使用这个`SGD`类，可以按如下方式进行神经网络的参数的更新(下面的代码是不能实际运行的伪代码)。

```
network = TwoLayerNet(...)
optimizer = SGD()

for i in range(10000):
    ...
    x_batch, t_batch = get_mini_batch(...) # mini-batch
    grads = network.gradient(x_batch, t_batch)
    params = network.params
    optimizer.update(params, grads)
    ...
```

这里首次出现的变量名`optimizer`表示“进行最优化的人”的意思，这里由SGD承担这个角色。参数的更新由`optimizer`负责完成。我们在这里需要做的只是将参数和梯度的信息传给`optimizer`。

像这样，通过单独实现进行最优化的类，功能的模块化变得更简单。比如，后面我们马上会实现另一个最优化方法Momentum，它同样会实现成拥有`update(params, grads)`这个共同方法的形式。这样一来，只需要将

`optimizer = SGD()`这一语句换成`optimizer = Momentum()`，就可以从SGD切换为Momentum。

很多深度学习框架都实现了各种最优化方法，并且提供了可以简单切换这些方法的构造。比如Lasagne深度学习框架，在`updates.py`这个文件中以函数的形式集中实现了最优化方法。用户可以从中选择自己想用的最优化方法。

6.1.3　SGD的缺点

虽然SGD简单，并且容易实现，但是在解决某些问题时可能没有效率。这里，在指出SGD的缺点之际，我们来思考一下求下面这个函数的最小值的问题。

$$f(x,y)=\frac{1}{20}x^2+y^2 \tag{6.2}$$

如图6-1所示，式(6.2)表示的函数是向x轴方向延伸的“碗”状函数。实际上，式(6.2)的等高线呈向x轴方向延伸的椭圆状。

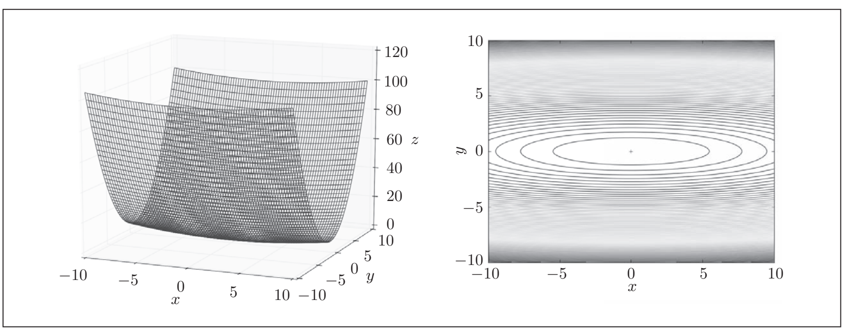


图6-1　$f(x,y)=\frac{1}{20}x^2+y^2$的图形（左图）和它的等高线（右图）

现在看一下式(6.2)表示的函数的梯度。如果用图表示梯度的话，则如图6-2所示。这个梯度的特征是，y轴方向上大，x轴方向上小。换句话说，就是y轴方向的坡度大，而x轴方向的坡度小。这里需要注意的是，虽然式(6.2)的最小值在$(x, y) = (0, 0)$处，但是图6-2中的梯度在很多地方并没有指向$(0, 0)$。

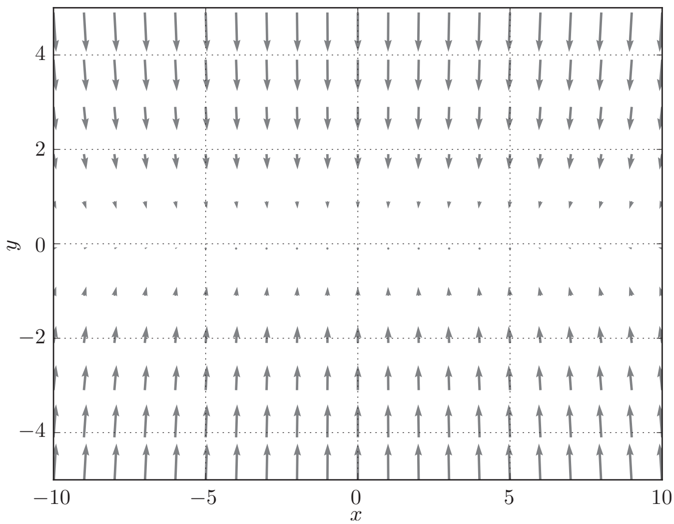


图6-2　$f(x, y) = \frac{1}{20}x^2 + y^2$ 的梯度

我们来尝试对图6-1这种形状的函数应用SGD。从$(x, y) = (-7.0, 2.0)$处(初始值)开始搜索，结果如图6-3所示。

在图6-3中，SGD呈“之”字形移动。这是一个相当低效的路径。也就是说，SGD的缺点是，如果函数的形状非均向(anisotropic)，比如呈延伸状，搜索的路径就会非常低效。因此，我们需要比单纯朝梯度方向前进的SGD更聪明的方法。SGD低效的根本原因是，梯度的方向并没有指向最小值的方向。

为了改正SGD的缺点，下面我们将介绍Momentum、AdaGrad、Adam这3种方法来取代SGD。我们会简单介绍各个方法，并用数学式和Python进行实现。

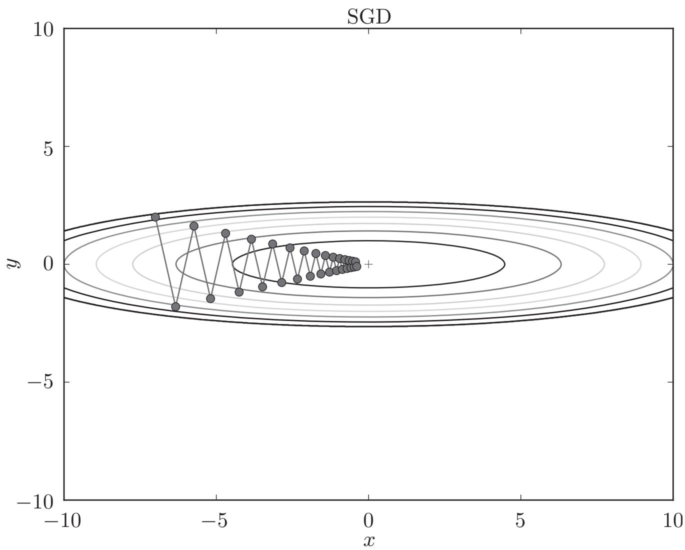


图6-3 基于SGD的最优化的更新路径：呈“之”字形朝最小值 $(0, 0)$ 移动，效率低

6.1.4 Momentum

Momentum是“动量”的意思，和物理有关。用数学式表示Momentum方法，如下所示。

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \eta \frac{\partial L}{\partial \boldsymbol{W}} \tag{6.3}$$

$$\boldsymbol{W} \leftarrow \boldsymbol{W} + \boldsymbol{v} \tag{6.4}$$

和前面的SGD一样，$\boldsymbol{W}$表示要更新的权重参数，$\frac{\partial L}{\partial \boldsymbol{W}}$表示损失函数关于$\boldsymbol{W}$的梯度，$\eta$表示学习率。这里新出现了一个变量$\boldsymbol{v}$，对应物理上的速度。式(6.3)表示了物体在梯度方向上受力，在这个力的作用下，物体的速度增加这一物理法则。如图6-4所示，Momentum方法给人的感觉就像是小球在地面上滚动。

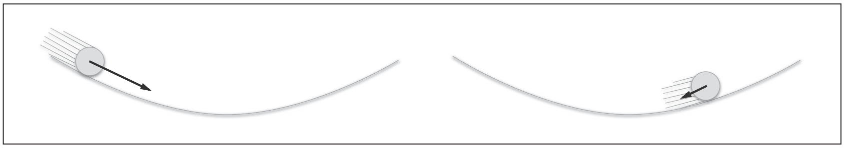


图6-4 Momentum：小球在斜面上滚动

式(6.3)中有$\alpha \boldsymbol{v}$这一项。在物体不受任何力时，该项承担使物体逐渐减速的任务（α设定为0.9之类的值），对应物理上的地面摩擦或空气阻力。下面是Momentum的代码实现（源代码在common/optimizer.py中）。

```
class Momentum:
    def __init__(self, lr=0.01, momentum=0.9):
        self.lr = lr
        self.momentum = momentum
        self.v = None

    def update(self, params, grads):
        if self.v is None:
            self.v = {}
            for key, val in params.items():
                self.v[key] = np.zeros_like(val)

        for key in params.keys():
            self.v[key] = self.momentum*self.v[key] - self.lr*grads[key]
            params[key] += self.v[key]
```

实例变量v会保存物体的速度。初始化时，v中什么都不保存，但当第一次调用update()时，v会以字典型变量的形式保存与参数结构相同的数据。剩余的代码部分就是将式(6.3)、式(6.4)写出来，很简单。

现在尝试使用Momentum解决式(6.2)的最优化问题，如图6-5所示。

图6-5中，更新路径就像小球在碗中滚动一样。和SGD相比，我们发现"之"字形的"程度"减轻了。这是因为虽然x轴方向上受到的力非常小，但是一直在同一方向上受力，所以朝同一个方向会有一定的加速。反过来，虽然y轴方向上受到的力很大，但是因为交互地受到正方向和反方向的力，它们会互相抵消，所以y轴方向上的速度不稳定。因此，和SGD时的情形相比，可以更快地朝x轴方向靠近，减弱"之"字形的变动程度。

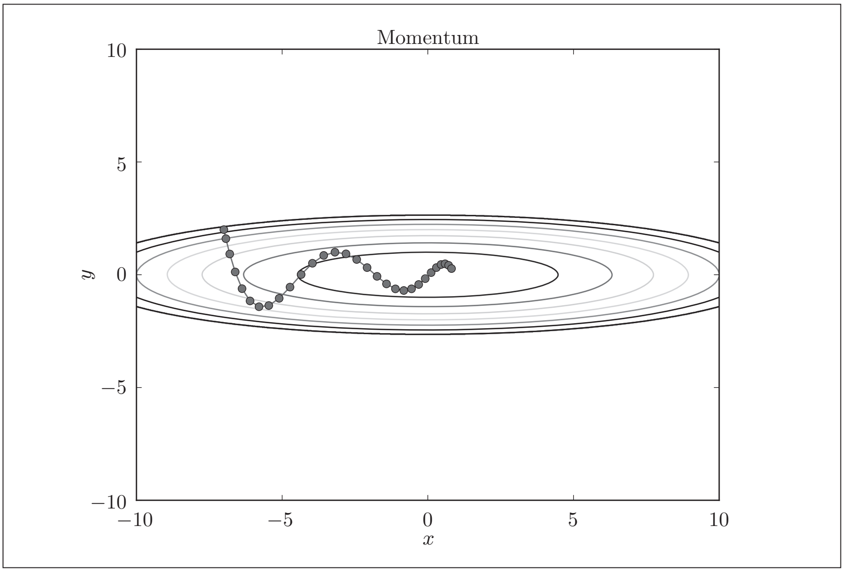


图6-5　基于Momentum的最优化的更新路径

6.1.5　AdaGrad

在神经网络的学习中，学习率（数学式中记为η）的值很重要。学习率过小，会导致学习花费过多时间；反过来，学习率过大，则会导致学习发散而不能正确进行。

在关于学习率的有效技巧中，有一种被称为**学习率衰减**（learning rate decay）的方法，即随着学习的进行，使学习率逐渐减小。实际上，一开始“多”学，然后逐渐“少”学的方法，在神经网络的学习中经常被使用。

逐渐减小学习率的想法，相当于将“全体”参数的学习率值一起降低。而AdaGrad[6]进一步发展了这个想法，针对“一个一个”的参数，赋予其“定制”的值。

AdaGrad会为参数的每个元素适当地调整学习率，与此同时进行学习（AdaGrad的Ada来自英文单词Adaptive，即“适当的”的意思）。下面，让我们用数学式表示AdaGrad的更新方法。

$$\boldsymbol{h} \leftarrow \boldsymbol{h} + \frac{\partial L}{\partial \boldsymbol{W}} \odot \frac{\partial L}{\partial \boldsymbol{W}} \tag{6.5}$$

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{1}{\sqrt{\boldsymbol{h}}} \frac{\partial L}{\partial \boldsymbol{W}} \tag{6.6}$$

和前面的SGD一样，$\boldsymbol{W}$表示要更新的权重参数，$\frac{\partial L}{\partial \boldsymbol{W}}$表示损失函数关于$\boldsymbol{W}$的梯度，$\eta$表示学习率。这里新出现了变量$\boldsymbol{h}$，如式(6.5)所示，它保存了以前的所有梯度值的平方和(式(6.5)中的$\odot$表示对应矩阵元素的乘法)。然后，在更新参数时，通过乘以$\frac{1}{\sqrt{\boldsymbol{h}}}$，就可以调整学习的尺度。这意味着，参数的元素中变动较大(被大幅更新)的元素的学习率将变小。也就是说，可以按参数的元素进行学习率衰减，使变动大的参数的学习率逐渐减小。

AdaGrad会记录过去所有梯度的平方和。因此，学习越深入，更新的幅度就越小。实际上，如果无止境地学习，更新量就会变为0，完全不再更新。为了改善这个问题，可以使用RMSProp [7] 方法。RMSProp方法并不是将过去所有的梯度一视同仁地相加，而是逐渐地遗忘过去的梯度，在做加法运算时将新梯度的信息更多地反映出来。这种操作从专业上讲，称为“指数移动平均”，呈指数函数式地减小过去的梯度的尺度。

现在来实现AdaGrad。AdaGrad的实现过程如下所示(源代码在`common/optimizer.py`中)。

```
class AdaGrad:
    def __init__(self, lr=0.01):
        self.lr = lr
        self.h = None

    def update(self, params, grads):
        if self.h is None:
            self.h = {}
            for key, val in params.items():
                self.h[key] = np.zeros_like(val)

    for key in params.keys():
        self.h[key] += grads[key] * grads[key]
        params[key] -= self.lr * grads[key] / (np.sqrt(self.h[key]) + 1e-7)
```

这里需要注意的是，最后一行加上了微小值`1e-7`。这是为了防止当`self.h[key]`中有0时，将0用作除数的情况。在很多深度学习的框架中，这个微小值也可以设定为参数，但这里我们用的是`1e-7`这个固定值。

现在，让我们试着使用AdaGrad解决式(6.2)的最优化问题，结果如图6-6所示。

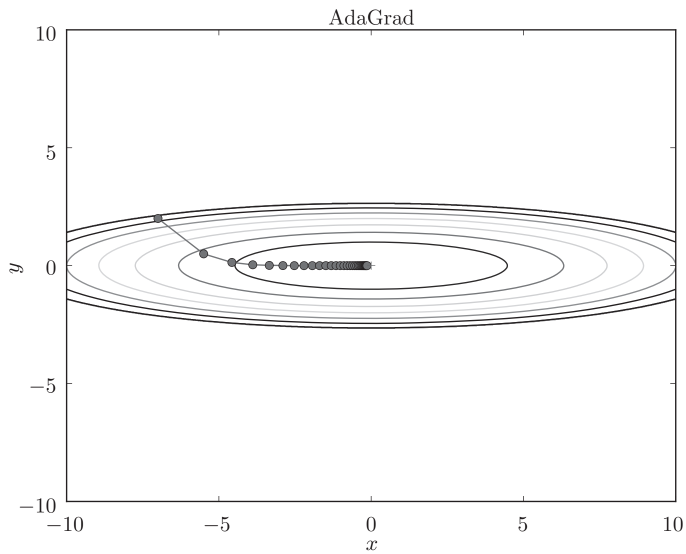


图6-6　基于AdaGrad的最优化的更新路径

由图6-6的结果可知，函数的取值高效地向着最小值移动。由于y轴方向上的梯度较大，因此刚开始变动较大，但是后面会根据这个较大的变动按比例进行调整，减小更新的步伐。因此，y轴方向上的更新程度被减弱，“之”字形的变动程度有所衰减。

6.1.6　Adam

Momentum参照小球在碗中滚动的物理规则进行移动，AdaGrad为参数的每个元素适当地调整更新步伐。如果将这两个方法融合在一起会怎么样

呢？这就是Adam[8]方法的基本思路①。

Adam是2015年提出的新方法。它的理论有些复杂，直观地讲，就是融合了Momentum和AdaGrad的方法。通过组合前面两个方法的优点，有望实现参数空间的高效搜索。此外，进行超参数的"偏置校正"也是Adam的特征。这里不再进行过多的说明，详细内容请参考原作者的论文[8]。关于Python的实现，`common/optimizer.py`中将其实现为了`Adam`类，有兴趣的读者可以参考。

现在，我们试着使用Adam解决式(6.2)的最优化问题，结果如图6-7所示。

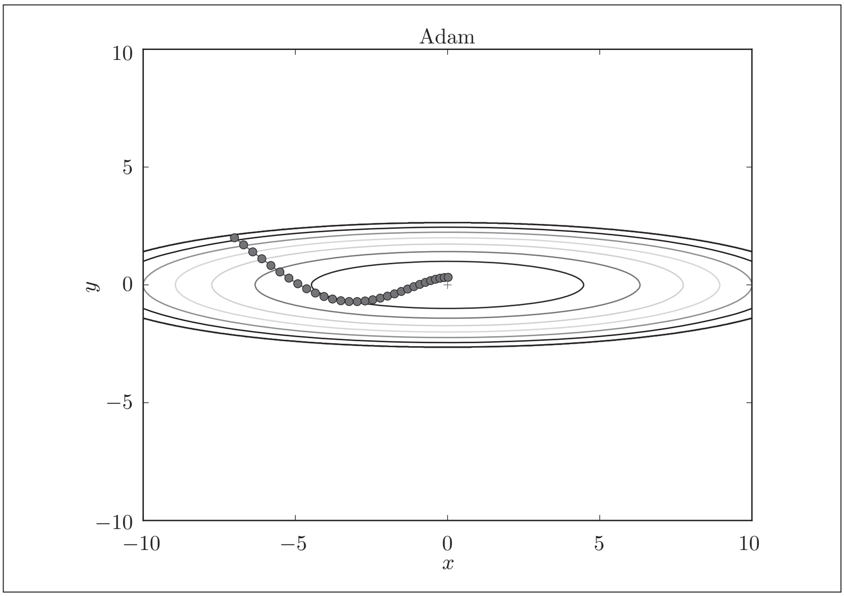


图6-7　基于Adam的最优化的更新路径

在图6-7中，基于Adam的更新过程就像小球在碗中滚动一样。虽然Momentun也有类似的移动，但是相比之下，Adam的小球左右摇晃的程度有所减轻。这得益于学习的更新程度被适当地调整了。

① 这里关于Adam方法的说明只是一个直观的说明，并不完全正确。详细内容请参考原作者的论文。

Adam会设置3个超参数。一个是学习率(论文中以α出现)，另外两个是一次momentum系数β_1和二次momentum系数β_2。根据论文，标准的设定值是β_1为0.9，β_2为0.999。设置了这些值后，大多数情况下都能顺利运行。

6.1.7　使用哪种更新方法呢

到目前为止，我们已经学习了4种更新参数的方法。这里我们来比较一下这4种方法(源代码在`ch06/optimizer_compare_naive.py`中)。

如图6-8所示，根据使用的方法不同，参数更新的路径也不同。只看这个图的话，AdaGrad似乎是最好的，不过也要注意，结果会根据要解决的问题而变。并且，很显然，超参数(学习率等)的设定值不同，结果也会发生变化。

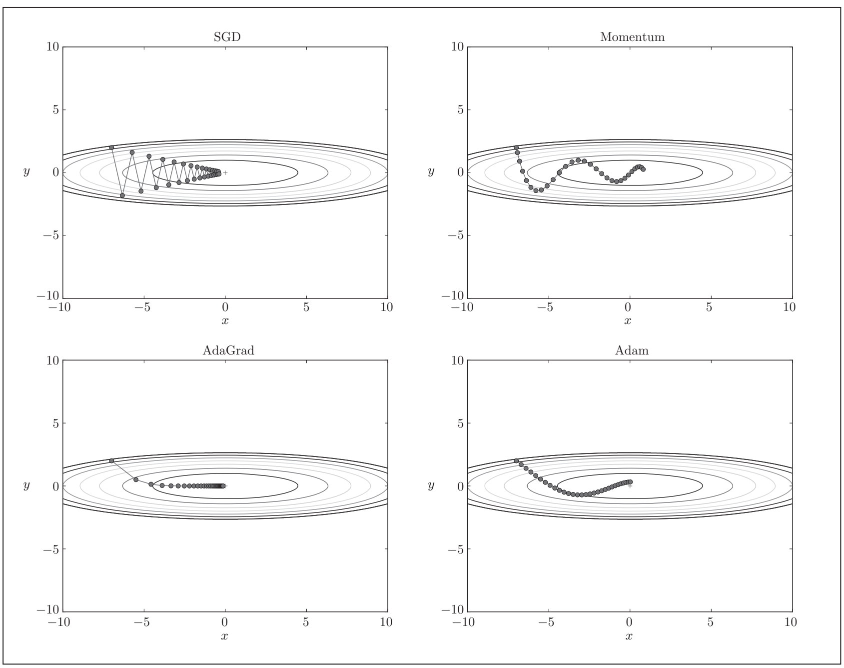


图6-8　最优化方法的比较：SGD、Momentum、AdaGrad、Adam

上面我们介绍了SGD、Momentum、AdaGrad、Adam这4种方法，那么用哪种方法好呢？非常遗憾，（目前）并不存在能在所有问题中都表现良好的方法。这4种方法各有各的特点，都有各自擅长解决的问题和不擅长解决的问题。

很多研究中至今仍在使用SGD。Momentum和AdaGrad也是值得一试的方法。最近，很多研究人员和技术人员都喜欢用Adam。本书将主要使用SGD或者Adam，读者可以根据自己的喜好多多尝试。

6.1.8 基于MNIST数据集的更新方法的比较

我们以手写数字识别为例，比较前面介绍的SGD、Momentum、AdaGrad、Adam这4种方法，并确认不同的方法在学习进展上有多大程度的差异。先来看一下结果，如图6-9所示（源代码在`ch06/optimizer_compare_mnist.py`中）。

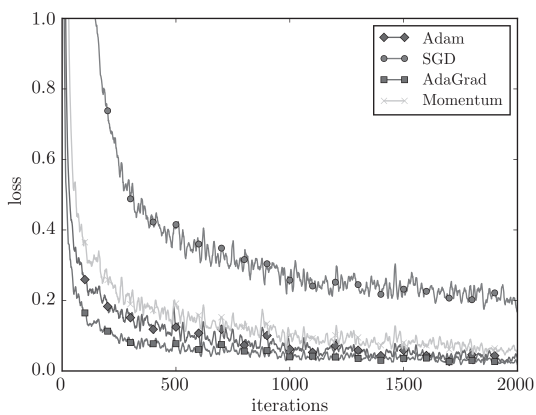


图6-9 基于MNIST数据集的4种更新方法的比较：横轴表示学习的迭代次数（iteration），纵轴表示损失函数的值（loss）

这个实验以一个5层神经网络为对象，其中每层有100个神经元。激活函数使用的是ReLU。

从图6-9的结果中可知，与SGD相比，其他3种方法学习得更快，而且速度基本相同，仔细看的话，AdaGrad的学习进行得稍微快一点。这个实验需要注意的地方是，实验结果会随学习率等超参数、神经网络的结构（几层深等）的不同而发生变化。不过，一般而言，与SGD相比，其他3种方法可以学习得更快，有时最终的识别精度也更高。

6.2 权重的初始值

在神经网络的学习中，权重的初始值特别重要。实际上，设定什么样的权重初始值，经常关系到神经网络的学习能否成功。本节将介绍权重初始值的推荐值，并通过实验确认神经网络的学习是否会快速进行。

6.2.1 可以将权重初始值设为0吗

后面我们会介绍抑制过拟合、提高泛化能力的技巧——权值衰减（weight decay）。简单地说，权值衰减就是一种以减小权重参数的值为目的进行学习的方法。通过减小权重参数的值来抑制过拟合的发生。

如果想减小权重的值，一开始就将初始值设为较小的值才是正途。实际上，在这之前的权重初始值都是像`0.01 * np.random.randn(10, 100)`这样，使用由高斯分布生成的值乘以0.01后得到的值（标准差为0.01的高斯分布）。

如果我们把权重初始值全部设为0以减小权重的值，会怎么样呢？从结论来说，将权重初始值设为0不是一个好主意。事实上，将权重初始值设为0的话，将无法正确进行学习。

为什么不能将权重初始值设为0呢？严格地说，为什么不能将权重初始值设成一样的值呢？这是因为在误差反向传播法中，所有的权重值都会进行相同的更新。比如，在2层神经网络中，假设第1层和第2层的权重为0。这样一来，正向传播时，因为输入层的权重为0，所以第2层的神经元全部会被传递相同的值。第2层的神经元中全部输入相同的值，这意味着反向传播时第2层的权重全部都会进行相同的更新（回忆一下“乘法节点的反向传播”

的内容)。因此，权重被更新为相同的值，并拥有了对称的值(重复的值)。这使得神经网络拥有许多不同的权重的意义丧失了。为了防止"权重均一化"(严格地讲，是为了瓦解权重的对称结构)，必须随机生成初始值。

6.2.2 隐藏层的激活值的分布

观察隐藏层的激活值①(激活函数的输出数据)的分布，可以获得很多启发。这里，我们来做一个简单的实验，观察权重初始值是如何影响隐藏层的激活值的分布的。这里要做的实验是，向一个5层神经网络(激活函数使用sigmoid函数)传入随机生成的输入数据，用直方图绘制各层激活值的数据分布。这个实验参考了斯坦福大学的课程CS231n[5]。

进行实验的源代码在`ch06/weight_init_activation_histogram.py`中，下面展示部分代码。

```
import numpy as np
import matplotlib.pyplot as plt

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

x = np.random.randn(1000, 100) # 1000个数据
node_num = 100        # 各隐藏层的节点(神经元)数
hidden_layer_size = 5 # 隐藏层有5层
activations = {}      # 激活值的结果保存在这里

for i in range(hidden_layer_size):
    if i != 0:
        x = activations[i-1]

    w = np.random.randn(node_num, node_num) * 1

    z = np.dot(x, w)
    a = sigmoid(z)   # sigmoid函数
    activations[i] = a
```

① 这里我们将激活函数的输出数据称为"激活值"，但是有的文献中会将在层之间流动的数据也称为"激活值"。

这里假设神经网络有5层，每层有100个神经元。然后，用高斯分布随机生成1000个数据作为输入数据，并把它们传给5层神经网络。激活函数使用sigmoid函数，各层的激活值的结果保存在`activations`变量中。这个代码段中需要注意的是权重的尺度。虽然这次我们使用的是标准差为1的高斯分布，但实验的目的是通过改变这个尺度（标准差），观察激活值的分布如何变化。现在，我们将保存在`activations`中的各层数据画成直方图。

```
# 绘制直方图
for i, a in activations.items():
    plt.subplot(1, len(activations), i+1)
    plt.title(str(i+1) + "-layer")
    plt.hist(a.flatten(), 30, range=(0,1))
plt.show()
```

运行这段代码后，可以得到图6-10的直方图。

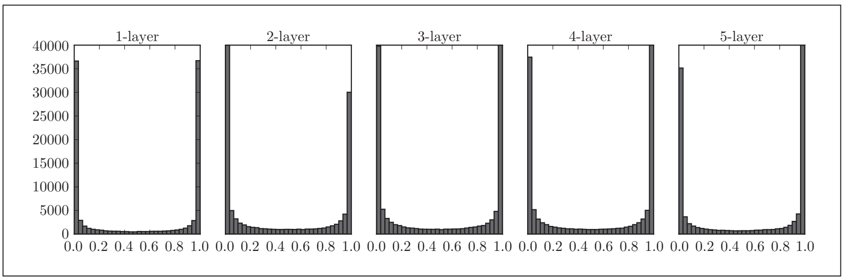


图6-10　使用标准差为1的高斯分布作为权重初始值时的各层激活值的分布

从图6-10可知，各层的激活值呈偏向0和1的分布。这里使用的sigmoid函数是S型函数，随着输出不断地靠近0（或者靠近1），它的导数的值逐渐接近0。因此，偏向0和1的数据分布会造成反向传播中梯度的值不断变小，最后消失。这个问题称为**梯度消失**（gradient vanishing）。层次加深的深度学习中，梯度消失的问题可能会更加严重。

下面，将权重的标准差设为0.01，进行相同的实验。实验的代码只需要把设定权重初始值的地方换成下面的代码即可。

```
# w = np.random.randn(node_num, node_num) * 1
w = np.random.randn(node_num, node_num) * 0.01
```

来看一下结果。使用标准差为0.01的高斯分布时，各层的激活值的分布如图6-11所示。

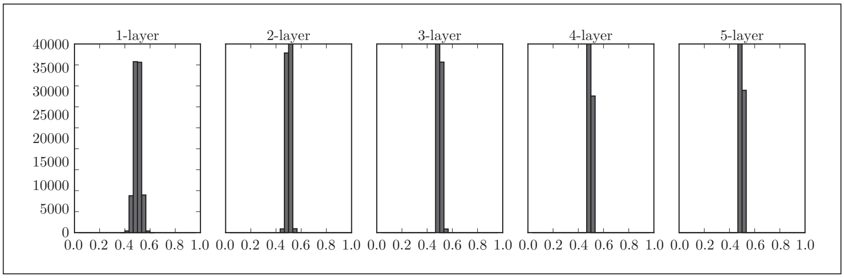


图6-11　使用标准差为0.01的高斯分布作为权重初始值时的各层激活值的分布

这次呈集中在0.5附近的分布。因为不像刚才的例子那样偏向0和1，所以不会发生梯度消失的问题。但是，激活值的分布有所偏向，说明在表现力上会有很大问题。为什么这么说呢？因为如果有多个神经元都输出几乎相同的值，那它们就没有存在的意义了。比如，如果100个神经元都输出几乎相同的值，那么也可以由1个神经元来表达基本相同的事情。因此，激活值在分布上有所偏向会出现“表现力受限”的问题。



各层的激活值的分布都要求有适当的广度。为什么呢？因为通过在各层间传递多样性的数据，神经网络可以进行高效的学习。反过来，如果传递的是有所偏向的数据，就会出现梯度消失或者“表现力受限”的问题，导致学习可能无法顺利进行。

接着，我们尝试使用Xavier Glorot等人的论文[9]中推荐的权重初始值（俗称“Xavier初始值”）。现在，在一般的深度学习框架中，Xavier初始值已被作为标准使用。比如，Caffe框架中，通过在设定权重初始值时赋予xavier参数，就可以使用Xavier初始值。

Xavier的论文中，为了使各层的激活值呈现出具有相同广度的分布，推

导了合适的权重尺度。推导出的结论是，如果前一层的节点数为n，则初始值使用标准差为 $\frac{1}{\sqrt{n}}$ 的分布[①]（图6-12）。

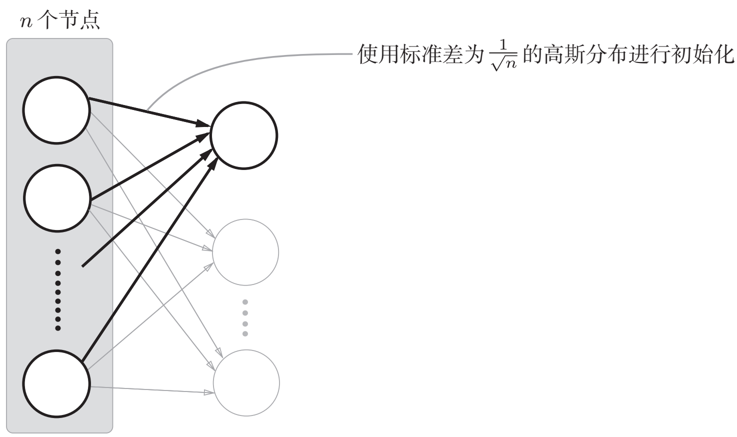


图6-12 Xavier初始值：与前一层有n个节点连接时，初始值使用标准差为 $\frac{1}{\sqrt{n}}$ 的分布

使用Xavier初始值后，前一层的节点数越多，要设定为目标节点的初始值的权重尺度就越小。现在，我们使用Xavier初始值进行实验。进行实验的代码只需要将设定权重初始值的地方换成如下内容即可（因为此处所有层的节点数都是100，所以简化了实现）。

```
node_num = 100 # 前一层的节点数
w = np.random.randn(node_num, node_num) / np.sqrt(node_num)
```

使用Xavier初始值后的结果如图6-13所示。从这个结果可知，越是后面的层，图像变得越歪斜，但是呈现了比之前更有广度的分布。因为各层间传递的数据有适当的广度，所以sigmoid函数的表现力不受限制，有望进行高效的学习。

① Xavier的论文中提出的设定值，不仅考虑了前一层的输入节点数量，还考虑了下一层的输出节点数量。但是，Caffe等框架的实现中进行了简化，只使用了这里所说的前一层的输入节点进行计算。

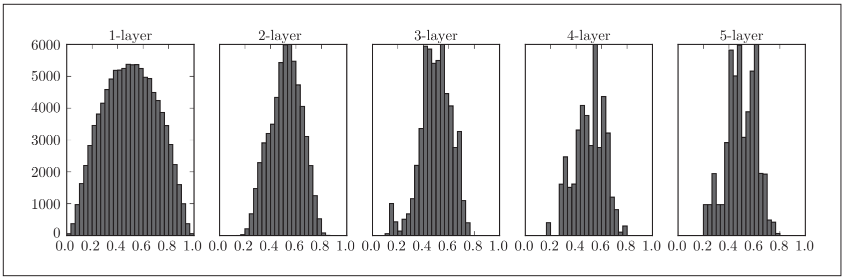


图6-13 使用Xavier初始值作为权重初始值时的各层激活值的分布



图6-13的分布中，后面的层的分布呈稍微歪斜的形状。如果用tanh函数（双曲线函数）代替sigmoid函数，这个稍微歪斜的问题就能得到改善。实际上，使用tanh函数后，会呈漂亮的吊钟型分布。tanh函数和sigmoid函数同是S型曲线函数，但tanh函数是关于原点(0, 0)对称的S型曲线，而sigmoid函数是关于$(x, y)=(0, 0.5)$对称的S型曲线。众所周知，用作激活函数的函数最好具有关于原点对称的性质。

6.2.3 ReLU的权重初始值

Xavier初始值是以激活函数是线性函数为前提而推导出来的。因为sigmoid函数和tanh函数左右对称，且中央附近可以视作线性函数，所以适合使用Xavier初始值。但当激活函数使用ReLU时，一般推荐使用ReLU专用的初始值，也就是Kaiming He等人推荐的初始值，也称为“He初始值”[10]。当前一层的节点数为n时，He初始值使用标准差为$\sqrt{\frac{2}{n}}$的高斯分布。当Xavier初始值是$\sqrt{\frac{1}{n}}$时，（直观上）可以解释为，因为ReLU的负值区域的值为0，为了使它更有广度，所以需要2倍的系数。

现在来看一下激活函数使用ReLU时激活值的分布。我们给出了3个实验的结果（图6-14），依次是权重初始值为标准差是0.01的高斯分布（下文简写为“std = 0.01”）时、初始值为Xavier初始值时、初始值为ReLU专用的“He初始值”时的结果。

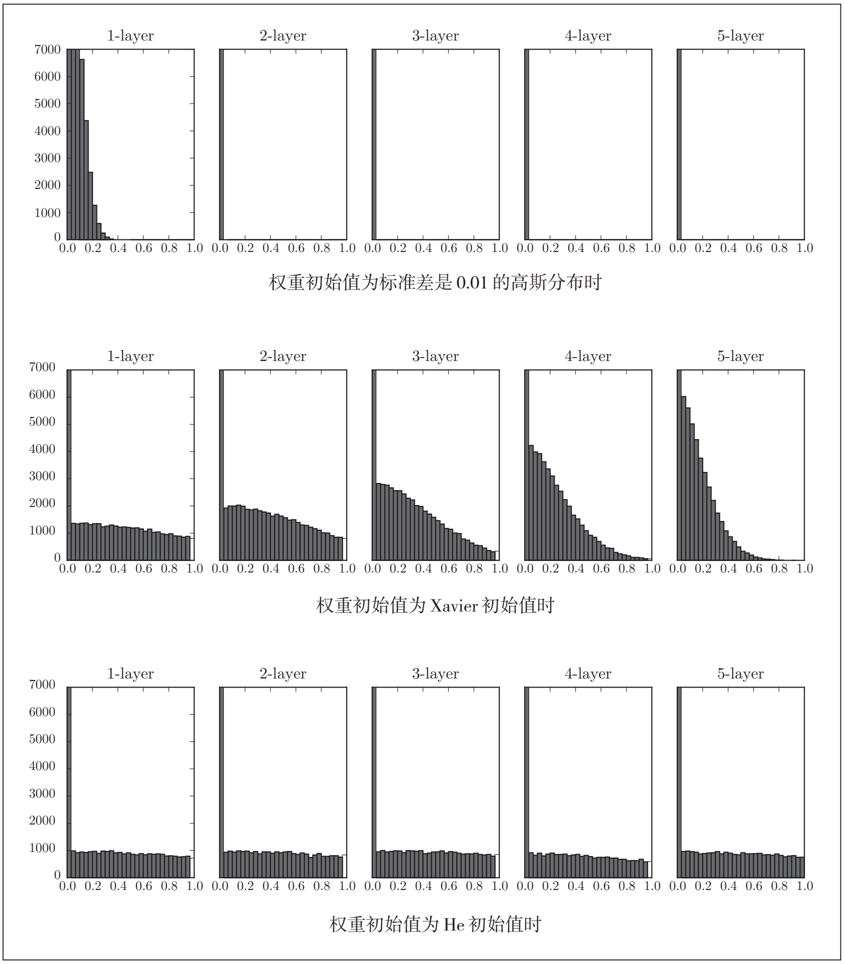


图6-14 激活函数使用ReLU时，不同权重初始值的激活值分布的变化

观察实验结果可知，当“std = 0.01”时，各层的激活值非常小[①]。神经网络上传递的是非常小的值，说明逆向传播时权重的梯度也同样很小。这是很严重的问题，实际上学习基本上没有进展。

① 各层激活值的分布平均值如下。1层：0.0396，2层：0.00290，3层：0.000197，4层：1.32e-5，5层：9.46e-7。

接下来是初始值为Xavier初始值时的结果。在这种情况下，随着层的加深，偏向一点点变大。实际上，层加深后，激活值的偏向变大，学习时会出现梯度消失的问题。而当初始值为He初始值时，各层中分布的广度相同。由于即便层加深，数据的广度也能保持不变，因此逆向传播时，也会传递合适的值。

总结一下，当激活函数使用ReLU时，权重初始值使用He初始值，当激活函数为`sigmoid`或`tanh`等S型曲线函数时，初始值使用Xavier初始值。这是目前的最佳实践。

6.2.4 基于MNIST数据集的权重初始值的比较

下面通过实际的数据，观察不同的权重初始值的赋值方法会在多大程度上影响神经网络的学习。这里，我们基于std = 0.01、Xavier初始值、He初始值进行实验（源代码在`ch06/weight_init_compare.py`中）。先来看一下结果，如图6-15所示。

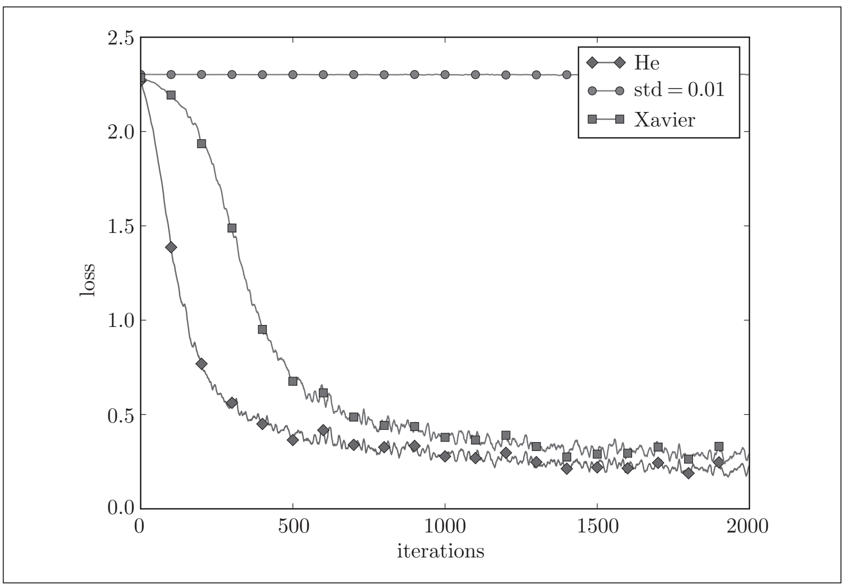


图6-15 基于MNIST数据集的权重初始值的比较：横轴是学习的迭代次数（iterations），纵轴是损失函数的值（loss）

这个实验中，神经网络有5层，每层有100个神经元，激活函数使用的是ReLU。从图6-15的结果可知，std = 0.01时完全无法进行学习。这和刚才观察到的激活值的分布一样，是因为正向传播中传递的值很小（集中在0附近的数据）。因此，逆向传播时求到的梯度也很小，权重几乎不进行更新。相反，当权重初始值为Xavier初始值和He初始值时，学习进行得很顺利。并且，我们发现He初始值时的学习进度更快一些。

综上，在神经网络的学习中，权重初始值非常重要。很多时候权重初始值的设定关系到神经网络的学习能否成功。权重初始值的重要性容易被忽视，而任何事情的开始（初始值）总是关键的，因此在结束本节之际，再次强调一下权重初始值的重要性。

6.3 Batch Normalization

在上一节，我们观察了各层的激活值分布，并从中了解到如果设定了合适的权重初始值，则各层的激活值分布会有适当的广度，从而可以顺利地进行学习。那么，为了使各层拥有适当的广度，“强制性”地调整激活值的分布会怎样呢？实际上，Batch Normalization[11]方法就是基于这个想法而产生的。

6.3.1 Batch Normalization 的算法

Batch Normalization（下文简称Batch Norm）是2015年提出的方法。Batch Norm虽然是一个问世不久的新方法，但已经被很多研究人员和技术人员广泛使用。实际上，看一下机器学习竞赛的结果，就会发现很多通过使用这个方法而获得优异结果的例子。

为什么Batch Norm这么惹人注目呢？因为Batch Norm有以下优点。

- 可以使学习快速进行（可以增大学习率）。
- 不那么依赖初始值（对于初始值不用那么神经质）。
- 抑制过拟合（降低Dropout等的必要性）。

考虑到深度学习要花费很多时间，第一个优点令人非常开心。另外，后两点也可以帮我们消除深度学习的学习中的很多烦恼。

如前所述，Batch Norm的思路是调整各层的激活值分布使其拥有适当的广度。为此，要向神经网络中插入对数据分布进行正规化的层，即Batch Normalization层（下文简称Batch Norm层），如图6-16所示。

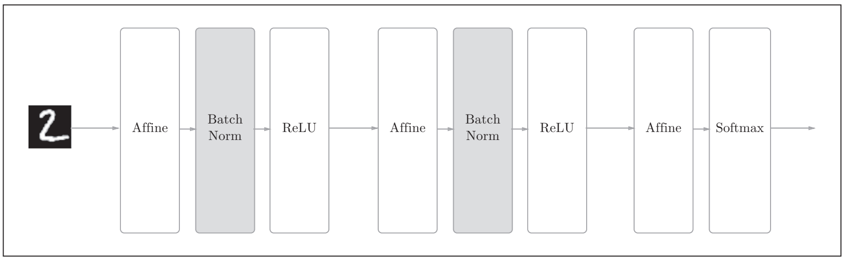


图6-16　使用了Batch Normalization的神经网络的例子（Batch Norm层的背景为灰色）

Batch Norm，顾名思义，以进行学习时的mini-batch为单位，按mini-batch进行正规化。具体而言，就是进行使数据分布的均值为0、方差为1的正规化。用数学式表示的话，如下所示。

$$\begin{aligned} \mu_B &\leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \\ \sigma_B^2 &\leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i-\mu_B)^2 \\ \hat{x}_i &\leftarrow \frac{x_i-\mu_B}{\sqrt{\sigma_B^2+\varepsilon}} \end{aligned} \tag{6.7}$$

这里对mini-batch的m个输入数据的集合$B=\{x_1,x_2,\cdots,x_m\}$求均值μ_B和方差σ_B^2。然后，对输入数据进行均值为0、方差为1（合适的分布）的正规化。式（6.7）中的ε是一个微小值（比如，`10e-7`等），它是为了防止出现除以0的情况。

式（6.7）所做的是将mini-batch的输入数据$\{x_1,x_2,\cdots,x_m\}$变换为均值

为0、方差为1的数据 $\{\hat{x}_1, \hat{x}_2, \cdots, \hat{x}_m\}$，非常简单。通过将这个处理插入到激活函数的前面（或者后面）①，可以减小数据分布的偏向。

接着，Batch Norm层会对正规化后的数据进行缩放和平移的变换，用数学式可以如下表示。

$$y_i \leftarrow \gamma \hat{x}_i + \beta \tag{6.8}$$

这里，γ和β是参数。一开始$\gamma = 1$，$\beta = 0$，然后再通过学习调整到合适的值。

上面就是Batch Norm的算法。这个算法是神经网络上的正向传播。如果使用第5章介绍的计算图，Batch Norm可以表示为图6-17。

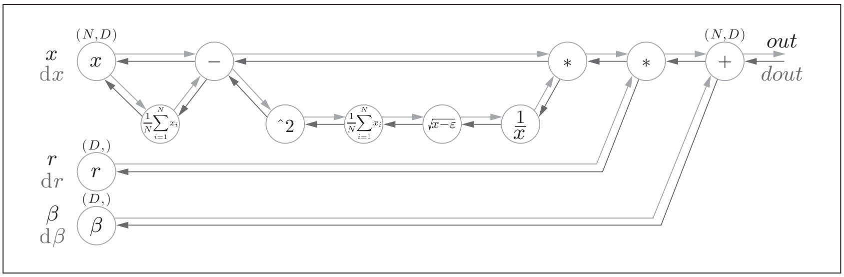


图6-17　Batch Normalization的计算图（引用自文献[13]）

Batch Norm的反向传播的推导有些复杂，这里我们不进行介绍。不过如果使用图6-17的计算图来思考的话，Batch Norm的反向传播或许也能比较轻松地推导出来。Frederik Kratzert 的博客“Understanding the backward pass through Batch Normalization Layer”[13]里有详细说明，感兴趣的读者可以参考一下。

6.3.2　Batch Normalization的评估

现在我们使用Batch Norm层进行实验。首先，使用MNIST数据集，

① 文献[11]、文献[12]等中有讨论（做过实验）应该把Batch Normalization插入到激活函数的前面还是后面。

观察使用Batch Norm层和不使用Batch Norm层时学习的过程会如何变化（源代码在`ch06/batch_norm_test.py`中），结果如图6-18所示。

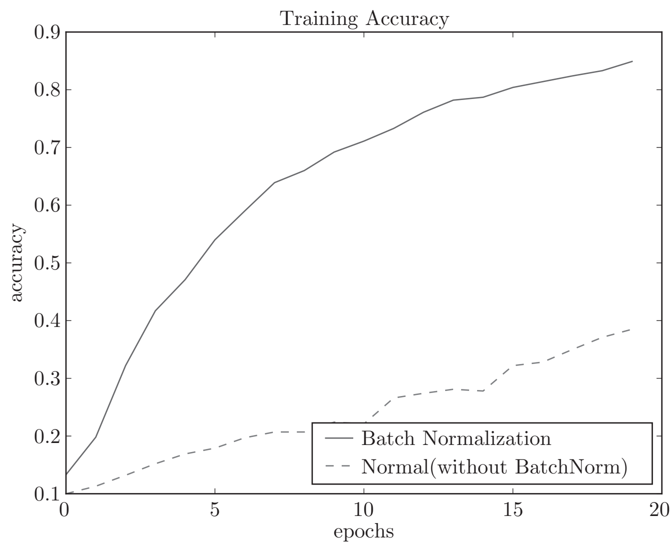


图6-18　基于Batch Norm的效果：使用Batch Norm后，学习进行得更快了

从图6-18的结果可知，使用Batch Norm后，学习进行得更快了。接着，给予不同的初始值尺度，观察学习的过程如何变化。图6-19是权重初始值的标准差为各种不同的值时的学习过程图。

我们发现，几乎所有的情况下都是使用Batch Norm时学习进行得更快。同时也可以发现，实际上，在不使用Batch Norm的情况下，如果不赋予一个尺度好的初始值，学习将完全无法进行。

综上，通过使用Batch Norm，可以推动学习的进行。并且，对权重初始值变得健壮（“对初始值健壮”表示不那么依赖初始值）。Batch Norm具备了如此优良的性质，一定能应用在更多场合中。

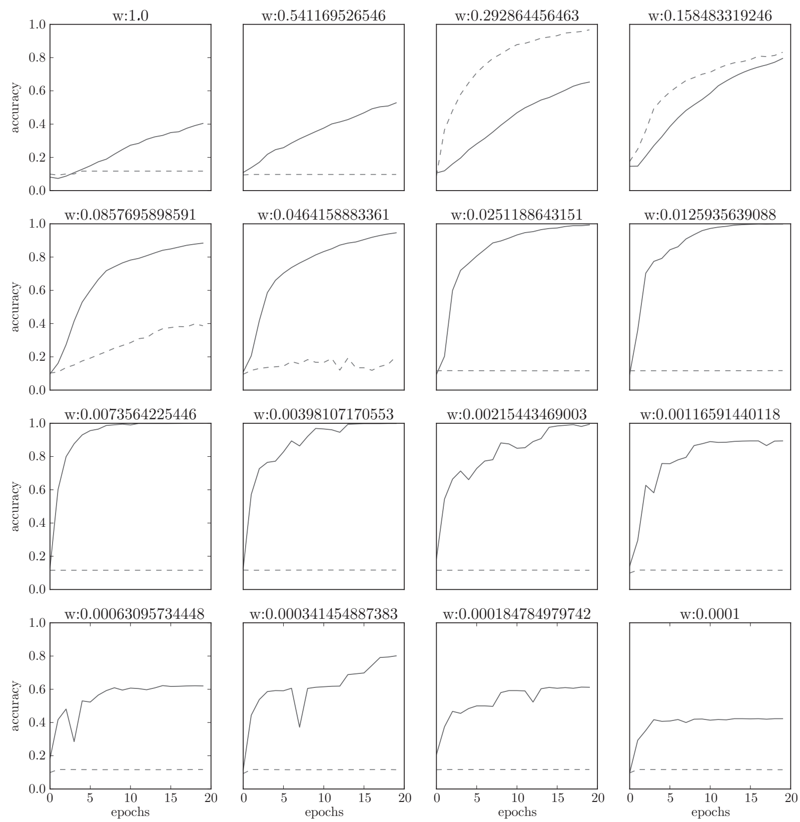


图6-19　图中的实线是使用了Batch Norm时的结果，虚线是没有使用Batch Norm时的结果：图的标题处标明了权重初始值的标准差

6.4　正则化

机器学习的问题中，**过拟合**是一个很常见的问题。过拟合指的是只能拟合训练数据，但不能很好地拟合不包含在训练数据中的其他数据的状态。机器学习的目标是提高泛化能力，即便是没有包含在训练数据里的未观测数据，也希望模型可以进行正确的识别。我们可以制作复杂的、表现力强的模型，

但是相应地，抑制过拟合的技巧也很重要。

6.4.1 过拟合

发生过拟合的原因，主要有以下两个。

- 模型拥有大量参数、表现力强。
- 训练数据少。

这里，我们故意满足这两个条件，制造过拟合现象。为此，要从MNIST数据集原本的60000个训练数据中只选定300个，并且，为了增加网络的复杂度，使用7层网络（每层有100个神经元，激活函数为ReLU）。

下面是用于实验的部分代码（对应文件在ch06/overfit_weight_decay.py中）。首先是用于读入数据的代码。

```
(x_train, t_train), (x_test, t_test) = load_mnist(normalize=True)
# 为了再现过拟合，减少学习数据
x_train = x_train[:300]
t_train = t_train[:300]
```

接着是进行训练的代码。和之前的代码一样，按epoch分别算出所有训练数据和所有测试数据的识别精度。

```
network = MultiLayerNet(input_size=784, hidden_size_list=[100, 100, 100,
100, 100, 100], output_size=10)
optimizer = SGD(lr=0.01) # 用学习率为0.01的SGD更新参数

max_epochs = 201
train_size = x_train.shape[0]
batch_size = 100

train_loss_list = []
train_acc_list = []
test_acc_list = []

iter_per_epoch = max(train_size / batch_size, 1)
epoch_cnt = 0

for i in range(1000000000):
    batch_mask = np.random.choice(train_size, batch_size)
```

```
        x_batch = x_train[batch_mask]
        t_batch = t_train[batch_mask]

        grads = network.gradient(x_batch, t_batch)
        optimizer.update(network.params, grads)

        if i % iter_per_epoch == 0:
            train_acc = network.accuracy(x_train, t_train)
            test_acc = network.accuracy(x_test, t_test)
            train_acc_list.append(train_acc)
            test_acc_list.append(test_acc)

            epoch_cnt += 1
            if epoch_cnt >= max_epochs:
                break
```

train_acc_list和test_acc_list中以epoch为单位（看完了所有训练数据的单位）保存识别精度。现在，我们将这些列表（train_acc_list、test_acc_list）绘成图，结果如图6-20所示。

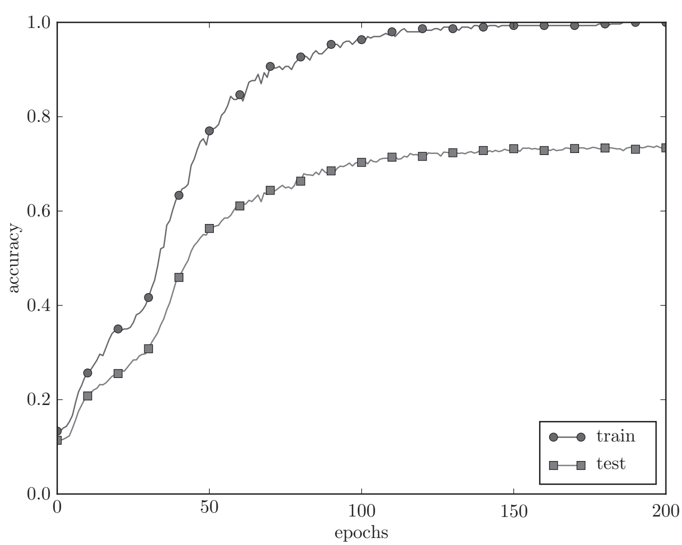


图6-20 训练数据（train）和测试数据（test）的识别精度的变化

过了100个epoch左右后，用训练数据测量到的识别精度几乎都为100%。但是，对于测试数据，离100%的识别精度还有较大的差距。如此大的识别精度差距，是只拟合了训练数据的结果。从图中可知，模型对训练时没有使用的一般数据（测试数据）拟合得不是很好。

6.4.2 权值衰减

权值衰减是一直以来经常被使用的一种抑制过拟合的方法。该方法通过在学习的过程中对大的权重进行惩罚，来抑制过拟合。很多过拟合原本就是因为权重参数取值过大才发生的。

复习一下，神经网络的学习目的是减小损失函数的值。这时，例如为损失函数加上权重的平方范数（L2范数）。这样一来，就可以抑制权重变大。用符号表示的话，如果将权重记为$\boldsymbol{W}$，L2范数的权值衰减就是$\frac{1}{2}\lambda\boldsymbol{W}^2$，然后将这个$\frac{1}{2}\lambda\boldsymbol{W}^2$加到损失函数上。这里，$\lambda$是控制正则化强度的超参数。$\lambda$设置得越大，对大的权重施加的惩罚就越重。此外，$\frac{1}{2}\lambda\boldsymbol{W}^2$开头的$\frac{1}{2}$是用于将$\frac{1}{2}\lambda\boldsymbol{W}^2$的求导结果变成$\lambda\boldsymbol{W}$的调整用常量。

对于所有权重，权值衰减方法都会为损失函数加上$\frac{1}{2}\lambda\boldsymbol{W}^2$。因此，在求权重梯度的计算中，要为之前的误差反向传播法的结果加上正则化项的导数$\lambda\boldsymbol{W}$。

L2范数相当于各个元素的平方和。用数学式表示的话，假设有权重$\boldsymbol{W} = (w_1, w_2, \cdots, w_n)$，则L2范数可用$\sqrt{w_1^2 + w_2^2 + \cdots + w_n^2}$计算出来。除了L2范数，还有L1范数、L∞范数等。L1范数是各个元素的绝对值之和，相当于$|w_1| + |w_2| + \cdots + |w_n|$。L∞范数也称为Max范数，相当于各个元素的绝对值中最大的那一个。L2范数、L1范数、L∞范数都可以用作正则化项，它们各有各的特点，不过这里我们要实现的是比较常用的L2范数。

现在我们来进行实验。对于刚刚进行的实验，应用$\lambda = 0.1$的权值衰减，结果如图6-21所示（对应权值衰减的网络在`common/multi_layer_net.py`中，用于实验的代码在`ch06/overfit_weight_decay.py`中）。

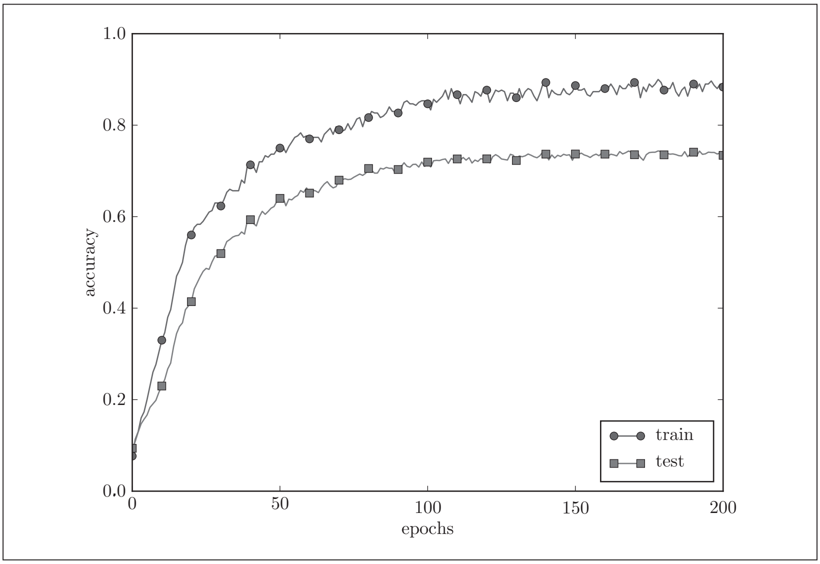


图6-21 使用了权值衰减的训练数据（train）和测试数据（test）的识别精度的变化

如图6-21所示，虽然训练数据的识别精度和测试数据的识别精度之间有差距，但是与没有使用权值衰减的图6-20的结果相比，差距变小了。这说明过拟合受到了抑制。此外，还要注意，训练数据的识别精度没有达到100%（1.0）。

6.4.3 Dropout

作为抑制过拟合的方法，前面我们介绍了为损失函数加上权重的L2范数的权值衰减方法。该方法可以简单地实现，在某种程度上能够抑制过拟合。但是，如果网络的模型变得很复杂，只用权值衰减就难以应对了。在这种情况下，我们经常会使用Dropout [14] 方法。

Dropout是一种在学习的过程中随机删除神经元的方法。训练时，随机选出隐藏层的神经元，然后将其删除。被删除的神经元不再进行信号的传递，如图6-22所示。训练时，每传递一次数据，就会随机选择要删除的神经元。然后，测试时，虽然会传递所有的神经元信号，但是对于各个神经元的输出，要乘上训练时的删除比例后再输出。

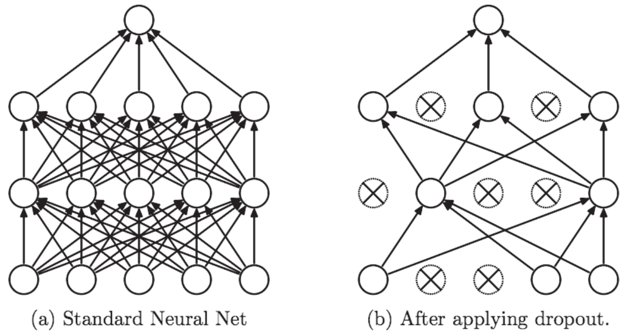


图6-22 Dropout的概念图（引用自文献[14]）：左边是一般的神经网络，右边是应用了Dropout的网络。Dropout通过随机选择并删除神经元，停止向前传递信号

下面我们来实现Dropout。这里的实现重视易理解性。不过，因为训练时如果进行恰当的计算的话，正向传播时单纯地传递数据就可以了（不用乘以删除比例），所以深度学习的框架中进行了这样的实现。关于高效的实现，可以参考Chainer中实现的Dropout。

```
class Dropout:
    def __init__(self, dropout_ratio=0.5):
        self.dropout_ratio = dropout_ratio
        self.mask = None

    def forward(self, x, train_flg=True):
        if train_flg:
            self.mask = np.random.rand(*x.shape) > self.dropout_ratio
            return x * self.mask
        else:
            return x * (1.0 - self.dropout_ratio)

    def backward(self, dout):
        return dout * self.mask
```

这里的要点是，每次正向传播时，`self.mask`中都会以`False`的形式保存要删除的神经元。`self.mask`会随机生成和`x`形状相同的数组，并将值比`dropout_ratio`大的元素设为`True`。反向传播时的行为和ReLU相同。也就是说，正向传播时传递了信号的神经元，反向传播时按原样传递信号；正向传播时

没有传递信号的神经元，反向传播时信号将停在那里。

现在，我们使用MNIST数据集进行验证，以确认Dropout的效果。源代码在`ch06/overfit_dropout.py`中。另外，源代码中使用了`Trainer`类来简化实现。

> `common/trainer.py`中实现了`Trainer`类。这个类可以负责前面所进行的网络的学习。详细内容可以参照`common/trainer.py`和`ch06/overfit_dropout.py`。

Dropout的实验和前面的实验一样，使用7层网络（每层有100个神经元，激活函数为ReLU），一个使用Dropout，另一个不使用Dropout，实验的结果如图6-23所示。

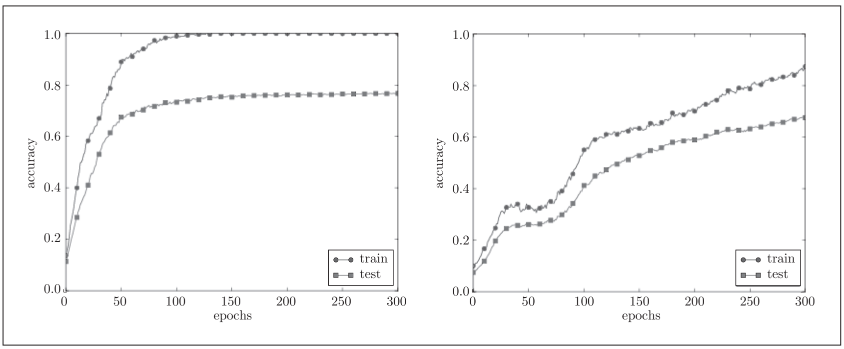


图6-23 左边没有使用Dropout，右边使用了Dropout（dropout_rate=0.15）

图6-23中，通过使用Dropout，训练数据和测试数据的识别精度的差距变小了。并且，训练数据也没有到达100%的识别精度。像这样，通过使用Dropout，即便是表现力强的网络，也可以抑制过拟合。

> 机器学习中经常使用集成学习。所谓集成学习，就是让多个模型单独进行学习，推理时再取多个模型的输出的平均值。用神经网络的语境来说，比如，准备5个结构相同（或者类似）的网络，分别进行学习，测试时，以这5个网络的输出的平均值作为答案。实验告诉我们，

通过进行集成学习，神经网络的识别精度可以提高好几个百分点。

这个集成学习与Dropout有密切的关系。这是因为可以将Dropout理解为，通过在学习过程中随机删除神经元，从而每一次都让不同的模型进行学习。并且，推理时，通过对神经元的输出乘以删除比例（比如，0.5等），可以取得模型的平均值。也就是说，可以理解成，Dropout将集成学习的效果（模拟地）通过一个网络实现了。

6.5 超参数的验证

神经网络中，除了权重和偏置等参数，**超参数**（hyper-parameter）也经常出现。这里所说的超参数是指，比如各层的神经元数量、batch大小、参数更新时的学习率或权值衰减等。如果这些超参数没有设置合适的值，模型的性能就会很差。虽然超参数的取值非常重要，但是在决定超参数的过程中一般会伴随很多的试错。本节将介绍尽可能高效地寻找超参数的值的方法。

6.5.1 验证数据

之前我们使用的数据集分成了训练数据和测试数据，训练数据用于学习，测试数据用于评估泛化能力。由此，就可以评估是否只过度拟合了训练数据（是否发生了过拟合），以及泛化能力如何等。

下面我们要对超参数设置各种各样的值以进行验证。这里要注意的是，不能使用测试数据评估超参数的性能。这一点非常重要，但也容易被忽视。

为什么不能用测试数据评估超参数的性能呢？这是因为如果使用测试数据调整超参数，超参数的值会对测试数据发生过拟合。换句话说，用测试数据确认超参数的值的“好坏”，就会导致超参数的值被调整为只拟合测试数据。这样的话，可能就会得到不能拟合其他数据、泛化能力低的模型。

因此，调整超参数时，必须使用超参数专用的确认数据。用于调整超参数的数据，一般称为**验证数据**（validation data）。我们使用这个验证数据来评估超参数的好坏。

训练数据用于参数（权重和偏置）的学习，验证数据用于超参数的性能评估。为了确认泛化能力，要在最后使用（比较理想的是只用一次）测试数据。

根据不同的数据集，有的会事先分成训练数据、验证数据、测试数据三部分，有的只分成训练数据和测试数据两部分，有的则不进行分割。在这种情况下，用户需要自行进行分割。如果是MNIST数据集，获得验证数据的最简单的方法就是从训练数据中事先分割20%作为验证数据，代码如下所示。

```
(x_train, t_train), (x_test, t_test) = load_mnist()

# 打乱训练数据
x_train, t_train = shuffle_dataset(x_train, t_train)

# 分割验证数据
validation_rate = 0.20
validation_num = int(x_train.shape[0] * validation_rate)

x_val = x_train[:validation_num]
t_val = t_train[:validation_num]
x_train = x_train[validation_num:]
t_train = t_train[validation_num:]
```

这里，分割训练数据前，先打乱了输入数据和教师标签。这是因为数据集的数据可能存在偏向（比如，数据从“0”到“10”按顺序排列等）。这里使用的 `shuffle_dataset` 函数利用了 `np.random.shuffle`，在 `common/util.py` 中有它的实现。

接下来，我们使用验证数据观察超参数的最优化方法。

6.5.2 超参数的最优化

进行超参数的最优化时，逐渐缩小超参数的“好值”的存在范围非常重要。所谓逐渐缩小范围，是指一开始先大致设定一个范围，从这个范围中随机选出一个超参数（采样），用这个采样到的值进行识别精度的评估；然后，多次重复该操作，观察识别精度的结果，根据这个结果缩小超参数的“好值”的范围。通过重复这一操作，就可以逐渐确定超参数的合适范围。

有报告[15]显示，在进行神经网络的超参数的最优化时，与网格搜索等有规律的搜索相比，随机采样的搜索方式效果更好。这是因为在多个超参数中，各个超参数对最终的识别精度的影响程度不同。

超参数的范围只要“大致地指定”就可以了。所谓“大致地指定”，是指像0.001（10^{-3}）到1000（10^3）这样，以“10的阶乘”的尺度指定范围（也表述为“用对数尺度（log scale）指定”）。

在超参数的最优化中，要注意的是深度学习需要很长时间（比如，几天或几周）。因此，在超参数的搜索中，需要尽早放弃那些不符合逻辑的超参数。于是，在超参数的最优化中，减少学习的epoch，缩短一次评估所需的时间是一个不错的办法。

以上就是超参数的最优化的内容，简单归纳一下，如下所示。

步骤0

设定超参数的范围。

步骤1

从设定的超参数范围中随机采样。

步骤2

使用步骤1中采样到的超参数的值进行学习，通过验证数据评估识别精度（但是要将epoch设置得很小）。

步骤3

重复步骤1和步骤2（100次等），根据它们的识别精度的结果，缩小超参数的范围。

反复进行上述操作，不断缩小超参数的范围，在缩小到一定程度时，从该范围中选出一个超参数的值。这就是进行超参数的最优化的一种方法。

这里介绍的超参数的最优化方法是实践性的方法。不过，这个方法与其说是科学方法，倒不如说有些实践者的经验的感觉。在超参数的最优化中，如果需要更精炼的方法，可以使用**贝叶斯最优化**（Bayesian optimization）。贝叶斯最优化运用以贝叶斯定理为中心的数学理论，能够更加严密、高效地进行最优化。详细内容请参考论文“Practical Bayesian Optimization of Machine Learning Algorithms”[16]等。

6.5.3 超参数最优化的实现

现在，我们使用MNIST数据集进行超参数的最优化。这里我们将学习率和控制权值衰减强度的系数（下文称为“权值衰减系数”）这两个超参数的搜索问题作为对象。这个问题的设定和解决思路参考了斯坦福大学的课程“CS231n”。

如前所述，通过从0.001（10^{-3}）到1000（10^{3}）这样的对数尺度的范围中随机采样进行超参数的验证。这在Python中可以写成`10 ** np.random.uniform(-3, 3)`。在该实验中，权值衰减系数的初始范围为10^{-8}到10^{-4}，学习率的初始范围为10^{-6}到10^{-2}。此时，超参数的随机采样的代码如下所示。

```
weight_decay = 10 ** np.random.uniform(-8, -4)
lr = 10 ** np.random.uniform(-6, -2)
```

像这样进行随机采样后，再使用那些值进行学习。之后，多次使用各种超参数的值重复进行学习，观察合乎逻辑的超参数在哪里。这里省略了具体实现，只列出了结果。进行超参数最优化的源代码在`ch06/hyperparameter_optimization.py`中，请大家自由参考。

下面我们就以权值衰减系数为10^{-8}到10^{-4}、学习率为10^{-6}到10^{-2}的范围进行实验，结果如图6-24所示。

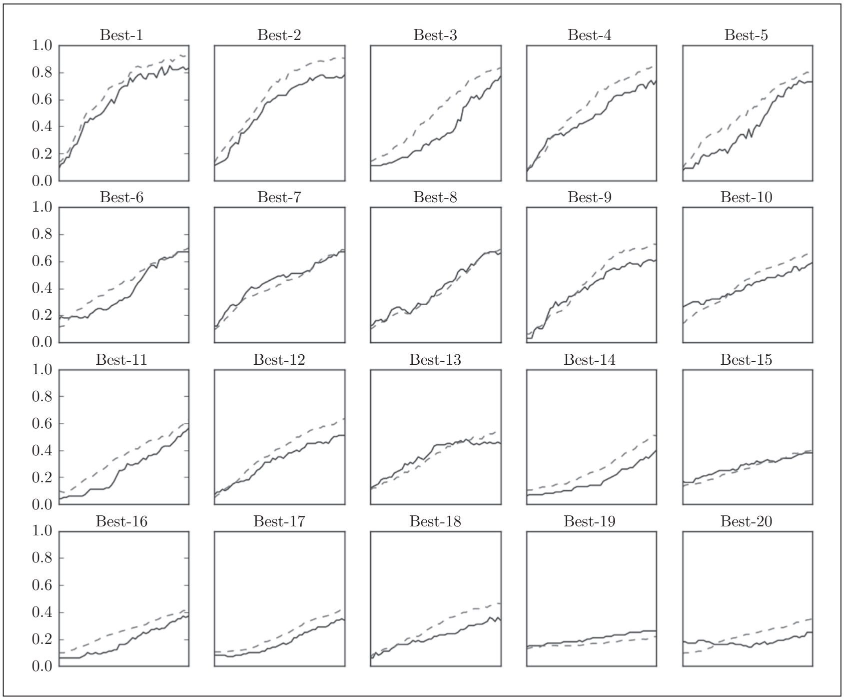


图6-24 实线是验证数据的识别精度，虚线是训练数据的识别精度

图6-24中，按识别精度从高到低的顺序排列了验证数据的学习的变化。从图中可知，直到“Best-5”左右，学习进行得都很顺利。因此，我们来观察一下“Best-5”之前的超参数的值（学习率和权值衰减系数），结果如下所示。

```
Best-1 (val acc:0.83) | lr:0.0092, weight decay:3.86e-07
Best-2 (val acc:0.78) | lr:0.00956, weight decay:6.04e-07
Best-3 (val acc:0.77) | lr:0.00571, weight decay:1.27e-06
Best-4 (val acc:0.74) | lr:0.00626, weight decay:1.43e-05
Best-5 (val acc:0.73) | lr:0.0052, weight decay:8.97e-06
```

从这个结果可以看出，学习率在0.001到0.01、权值衰减系数在10^{-8}到10^{-6}之间时，学习可以顺利进行。像这样，观察可以使学习顺利进行的超参数的范围，从而缩小值的范围。然后，在这个缩小的范围中重复相同的操作。

这样就能缩小到合适的超参数的存在范围，然后在某个阶段，选择一个最终的超参数的值。

6.6　小结

本章我们介绍了神经网络的学习中的几个重要技巧。参数的更新方法、权重初始值的赋值方法、Batch Normalization、Dropout等，这些都是现代神经网络中不可或缺的技术。另外，这里介绍的技巧，在最先进的深度学习中也被频繁使用。

本章所学的内容

- 参数的更新方法，除了SGD之外，还有Momentum、AdaGrad、Adam等方法。
- 权重初始值的赋值方法对进行正确的学习非常重要。
- 作为权重初始值，Xavier初始值、He初始值等比较有效。
- 通过使用Batch Normalization，可以加速学习，并且对初始值变得健壮。
- 抑制过拟合的正则化技术有权值衰减、Dropout等。
- 逐渐缩小“好值”存在的范围是搜索超参数的一个有效方法。

第7章
卷积神经网络

本章的主题是卷积神经网络（Convolutional Neural Network，**CNN**）。CNN被用于图像识别、语音识别等各种场合，在图像识别的比赛中，基于深度学习的方法几乎都以CNN为基础。本章将详细介绍CNN的结构，并用Python实现其处理内容。

7.1 整体结构

首先，来看一下CNN的网络结构，了解CNN的大致框架。CNN和之前介绍的神经网络一样，可以像乐高积木一样通过组装层来构建。不过，CNN中新出现了卷积层（Convolution层）和池化层（Pooling层）。卷积层和池化层将在下一节详细介绍，这里我们先看一下如何组装层以构建CNN。

之前介绍的神经网络中，相邻层的所有神经元之间都有连接，这称为**全连接**（fully-connected）。另外，我们用Affine层实现了全连接层。如果使用这个Affine层，一个5层的全连接的神经网络就可以通过图7-1所示的网络结构来实现。

如图7-1所示，全连接的神经网络中，Affine层后面跟着激活函数ReLU层（或者Sigmoid层）。这里堆叠了4层“Affine-ReLU”组合，然后第5层是Affine层，最后由Softmax层输出最终结果（概率）。

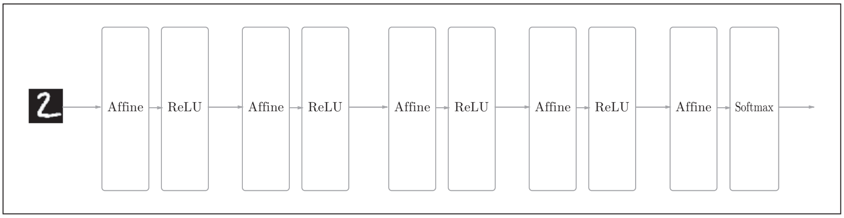


图7-1　基于全连接层（Affine层）的网络的例子

那么，CNN会是什么样的结构呢？图7-2是CNN的一个例子。

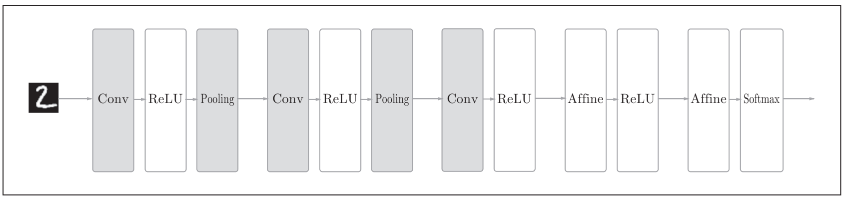


图7-2　基于CNN的网络的例子：新增了Convolution层和Pooling层（用灰色的方块表示）

如图7-2所示，CNN中新增了Convolution层和Pooling层。CNN的层的连接顺序是“Convolution - ReLU -（Pooling）”（Pooling层有时会被省略）。这可以理解为之前的“Affine - ReLU”连接被替换成了“Convolution - ReLU -（Pooling）”连接。

还需要注意的是，在图7-2的CNN中，靠近输出的层中使用了之前的“Affine - ReLU”组合。此外，最后的输出层中使用了之前的“Affine - Softmax”组合。这些都是一般的CNN中比较常见的结构。

7.2　卷积层

CNN中出现了一些特有的术语，比如填充、步幅等。此外，各层中传递的数据是有形状的数据（比如，3维数据），这与之前的全连接网络不同，因此刚开始学习CNN时可能会感到难以理解。本节我们将花点时间，认真学习一下CNN中使用的卷积层的结构。

7.2.1 全连接层存在的问题

之前介绍的全连接的神经网络中使用了全连接层（Affine层）。在全连接层中，相邻层的神经元全部连接在一起，输出的数量可以任意决定。

全连接层存在什么问题呢？那就是数据的形状被“忽视”了。比如，输入数据是图像时，图像通常是高、长、通道方向上的3维形状。但是，向全连接层输入时，需要将3维数据拉平为1维数据。实际上，前面提到的使用了MNIST数据集的例子中，输入图像就是1通道、高28像素、长28像素的（1, 28, 28）形状，但却被排成1列，以784个数据的形式输入到最开始的Affine层。

图像是3维形状，这个形状中应该含有重要的空间信息。比如，空间上邻近的像素为相似的值、RBG的各个通道之间分别有密切的关联性、相距较远的像素之间没有什么关联等，3维形状中可能隐藏有值得提取的本质模式。但是，因为全连接层会忽视形状，将全部的输入数据作为相同的神经元（同一维度的神经元）处理，所以无法利用与形状相关的信息。

而卷积层可以保持形状不变。当输入数据是图像时，卷积层会以3维数据的形式接收输入数据，并同样以3维数据的形式输出至下一层。因此，在CNN中，可以（有可能）正确理解图像等具有形状的数据。

另外，CNN中，有时将卷积层的输入输出数据称为**特征图**（feature map）。其中，卷积层的输入数据称为**输入特征图**（input feature map），输出数据称为**输出特征图**（output feature map）。本书中将“输入输出数据”和“特征图”作为含义相同的词使用。

7.2.2 卷积运算

卷积层进行的处理就是卷积运算。卷积运算相当于图像处理中的“滤波器运算”。在介绍卷积运算时，我们来看一个具体的例子（图7-3）。

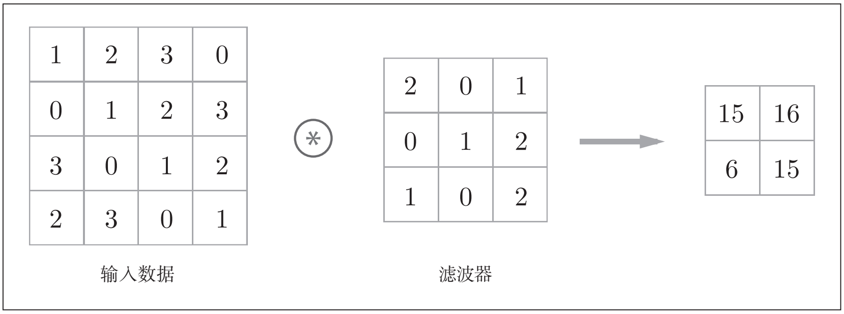


图7-3 卷积运算的例子：用“⊛”符号表示卷积运算

如图7-3所示，卷积运算对输入数据应用滤波器。在这个例子中，输入数据是有高长方向的形状的数据，滤波器也一样，有高长方向上的维度。假设用(height, width)表示数据和滤波器的形状，则在本例中，输入大小是(4, 4)，滤波器大小是(3, 3)，输出大小是(2, 2)。另外，有的文献中也会用“核”这个词来表示这里所说的“滤波器”。

现在来解释一下图7-3的卷积运算的例子中都进行了什么样的计算。图7-4中展示了卷积运算的计算顺序。

对于输入数据，卷积运算以一定间隔滑动滤波器的窗口并应用。这里所说的窗口是指图7-4中灰色的3×3的部分。如图7-4所示，将各个位置上滤波器的元素和输入的对应元素相乘，然后再求和(有时将这个计算称为**乘积累加运算**)。然后，将这个结果保存到输出的对应位置。将这个过程在所有位置都进行一遍，就可以得到卷积运算的输出。

在全连接的神经网络中，除了权重参数，还存在偏置。CNN中，滤波器的参数就对应之前的权重。并且，CNN中也存在偏置。图7-3的卷积运算的例子一直展示到了应用滤波器的阶段。包含偏置的卷积运算的处理流如图7-5所示。

如图7-5所示，向应用了滤波器的数据加上了偏置。偏置通常只有1个(1×1)(本例中，相对于应用了滤波器的4个数据，偏置只有1个)，这个值会被加到应用了滤波器的所有元素上。

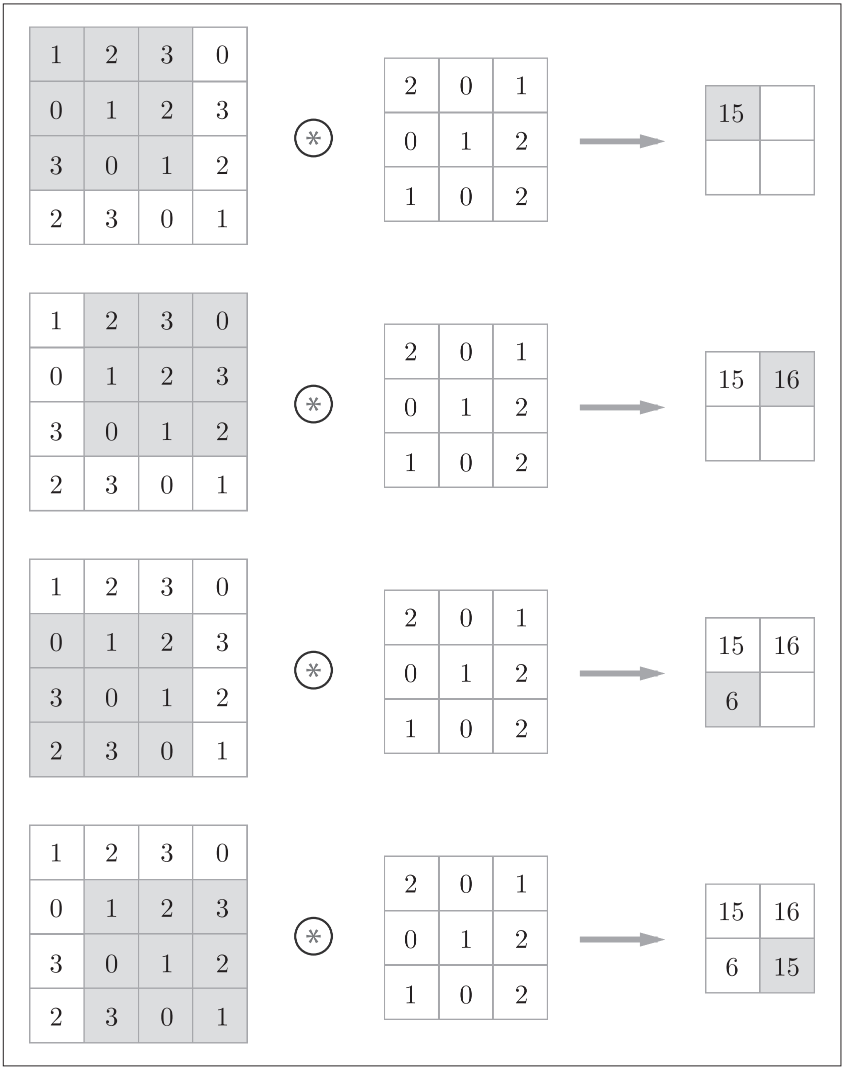


图 7-4 卷积运算的计算顺序

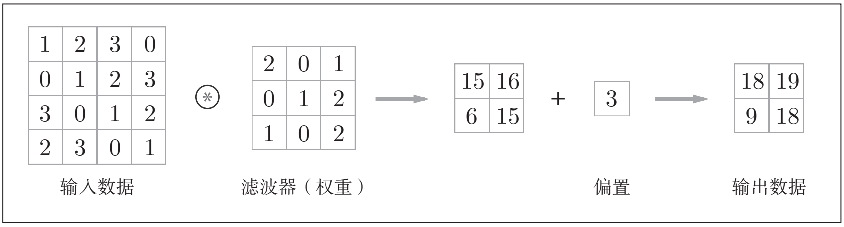


图7-5　卷积运算的偏置：向应用了滤波器的元素加上某个固定值（偏置）

7.2.3　填充

在进行卷积层的处理之前，有时要向输入数据的周围填入固定的数据（比如0等），这称为**填充**（padding），是卷积运算中经常会用到的处理。比如，在图7-6的例子中，对大小为(4, 4)的输入数据应用了幅度为1的填充。“幅度为1的填充”是指用幅度为1像素的0填充周围。

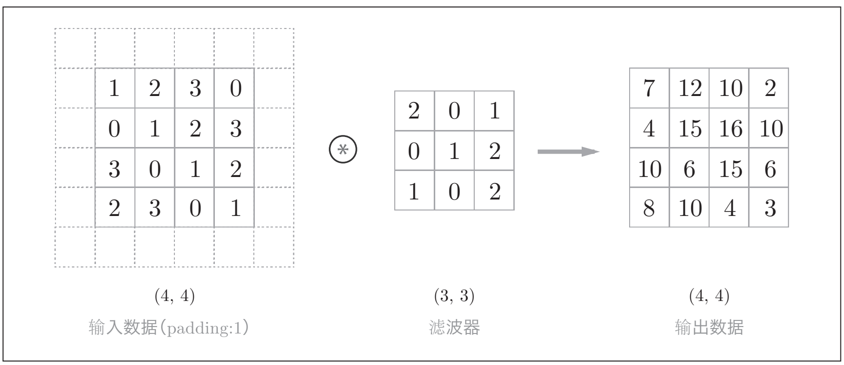


图7-6　卷积运算的填充处理：向输入数据的周围填入0（图中用虚线表示填充，并省略了填充的内容“0”）

如图7-6所示，通过填充，大小为(4, 4)的输入数据变成了(6, 6)的形状。然后，应用大小为(3, 3)的滤波器，生成了大小为(4, 4)的输出数据。这个例子中将填充设成了1，不过填充的值也可以设置成2、3等任意的整数。在图7-5的例子中，如果将填充设为2，则输入数据的大小变为(8, 8)；如果将填充设为3，则大小变为(10, 10)。

使用填充主要是为了调整输出的大小。比如，对大小为 $(4, 4)$ 的输入数据应用 $(3, 3)$ 的滤波器时，输出大小变为 $(2, 2)$，相当于输出大小比输入大小缩小了 2 个元素。这在反复进行多次卷积运算的深度网络中会成为问题。为什么呢？因为如果每次进行卷积运算都会缩小空间，那么在某个时刻输出大小就有可能变为 1，导致无法再应用卷积运算。为了避免出现这样的情况，就要使用填充。在刚才的例子中，将填充的幅度设为 1，那么相对于输入大小 $(4, 4)$，输出大小也保持为原来的 $(4, 4)$。因此，卷积运算就可以在保持空间大小不变的情况下将数据传给下一层。

7.2.4　步幅

应用滤波器的位置间隔称为**步幅**（stride）。之前的例子中步幅都是 1，如果将步幅设为 2，则如图 7-7 所示，应用滤波器的窗口的间隔变为 2 个元素。

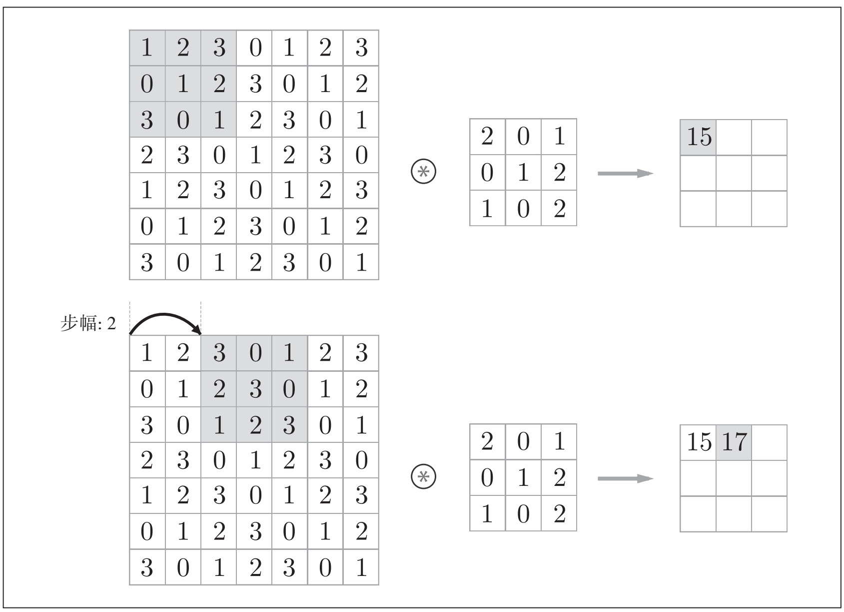


图 7-7　步幅为 2 的卷积运算的例子

在图7-7的例子中，对输入大小为$(7,7)$的数据，以步幅2应用了滤波器。通过将步幅设为2，输出大小变为$(3,3)$。像这样，步幅可以指定应用滤波器的间隔。

综上，增大步幅后，输出大小会变小。而增大填充后，输出大小会变大。如果将这样的关系写成算式，会如何呢？接下来，我们看一下对于填充和步幅，如何计算输出大小。

这里，假设输入大小为(H,W)，滤波器大小为(FH,FW)，输出大小为(OH,OW)，填充为P，步幅为S。此时，输出大小可通过式(7.1)进行计算。

$$
\begin{aligned}
OH &= \frac{H+2P-FH}{S}+1 \\
OW &= \frac{W+2P-FW}{S}+1
\end{aligned}
\tag{7.1}
$$

现在，我们使用这个算式，试着做几个计算。

例1：图7-6的例子

输入大小：$(4,4)$；填充：1；步幅：1；滤波器大小：$(3,3)$

$$
\begin{aligned}
OH &= \frac{4+2\cdot 1-3}{1}+1=4 \\
OW &= \frac{4+2\cdot 1-3}{1}+1=4
\end{aligned}
$$

例2：图7-7的例子

输入大小：$(7,7)$；填充：0；步幅：2；滤波器大小：$(3,3)$

$$
\begin{aligned}
OH &= \frac{7+2\cdot 0-3}{2}+1=3 \\
OW &= \frac{7+2\cdot 0-3}{2}+1=3
\end{aligned}
$$

例3

输入大小：$(28,31)$；填充：2；步幅：3；滤波器大小：$(5,5)$

$$OH = \frac{28 + 2 \cdot 2 - 5}{3} + 1 = 10$$
$$OW = \frac{31 + 2 \cdot 2 - 5}{3} + 1 = 11$$

如这些例子所示，通过在式(7.1)中代入值，就可以计算输出大小。这里需要注意的是，虽然只要代入值就可以计算输出大小，但是所设定的值必须使式(7.1)中的 $\frac{W+2P-FW}{S}$ 和 $\frac{H+2P-FH}{S}$ 分别可以除尽。当输出大小无法除尽时(结果是小数时)，需要采取报错等对策。顺便说一下，根据深度学习的框架的不同，当值无法除尽时，有时会向最接近的整数四舍五入，不进行报错而继续运行。

7.2.5 3维数据的卷积运算

之前的卷积运算的例子都是以有高、长方向的2维形状为对象的。但是，图像是3维数据，除了高、长方向之外，还需要处理通道方向。这里，我们按照与之前相同的顺序，看一下对加上了通道方向的3维数据进行卷积运算的例子。

图7-8是卷积运算的例子，图7-9是计算顺序。这里以3通道的数据为例，展示了卷积运算的结果。和2维数据时(图7-3的例子)相比，可以发现纵深方向(通道方向)上特征图增加了。通道方向上有多个特征图时，会按通道进行输入数据和滤波器的卷积运算，并将结果相加，从而得到输出。

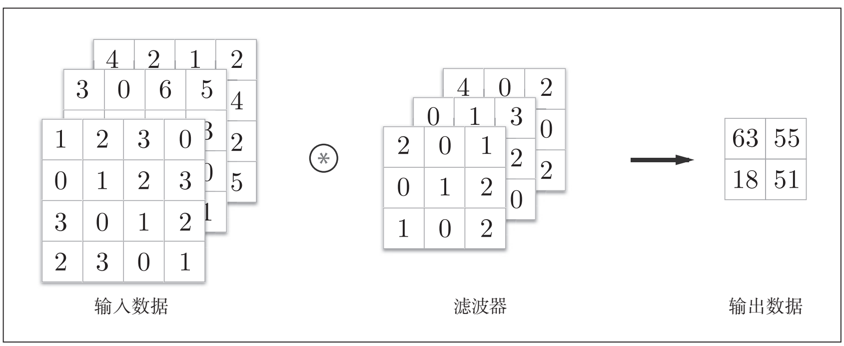


图7-8　对3维数据进行卷积运算的例子

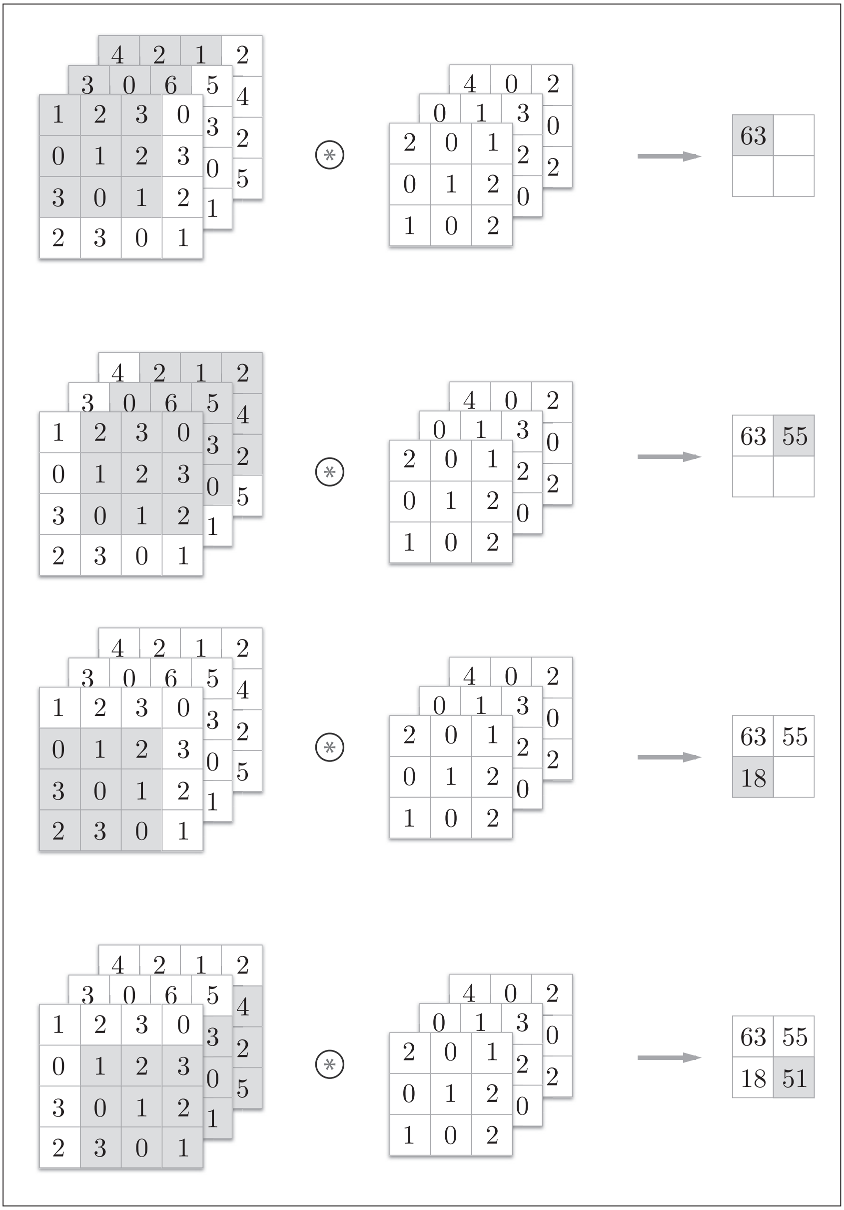


图7-9　对3维数据进行卷积运算的计算顺序

需要注意的是，在3维数据的卷积运算中，输入数据和滤波器的通道数要设为相同的值。在这个例子中，输入数据和滤波器的通道数一致，均为3。滤波器大小可以设定为任意值（不过，每个通道的滤波器大小要全部相同）。这个例子中滤波器大小为(3, 3)，但也可以设定为(2, 2)、(1, 1)、(5, 5)等任意值。再强调一下，通道数只能设定为和输入数据的通道数相同的值（本例中为3）。

7.2.6　结合方块思考

将数据和滤波器结合长方体的方块来考虑，3维数据的卷积运算会很容易理解。方块是如图7-10所示的3维长方体。把3维数据表示为多维数组时，书写顺序为（channel, height, width）。比如，通道数为C、高度为H、长度为W的数据的形状可以写成(C, H, W)。滤波器也一样，要按（channel, height, width）的顺序书写。比如，通道数为C、滤波器高度为FH（Filter Height）、长度为FW（Filter Width）时，可以写成(C, FH, FW)。

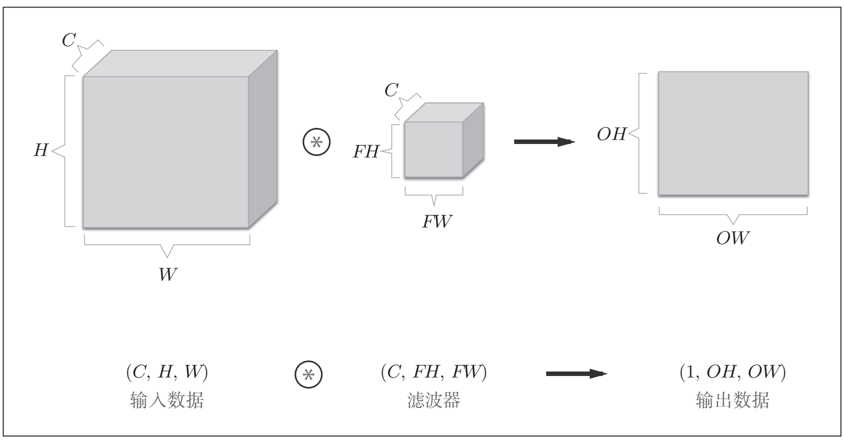


图7-10　结合方块思考卷积运算。请注意方块的形状

在这个例子中，数据输出是1张特征图。所谓1张特征图，换句话说，就是通道数为1的特征图。那么，如果要在通道方向上也拥有多个卷积运算

的输出，该怎么做呢？为此，就需要用到多个滤波器（权重）。用图表示的话，如图7-11所示。

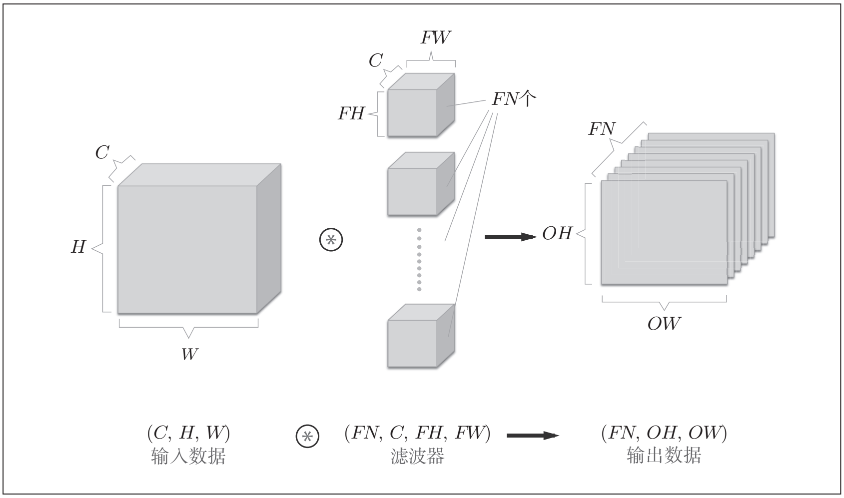


图7-11　基于多个滤波器的卷积运算的例子

图7-11中，通过应用*FN*个滤波器，输出特征图也生成了*FN*个。如果将这*FN*个特征图汇集在一起，就得到了形状为(*FN*, *OH*, *OW*)的方块。将这个方块传给下一层，就是CNN的处理流。

如图7-11所示，关于卷积运算的滤波器，也必须考虑滤波器的数量。因此，作为4维数据，滤波器的权重数据要按(output_channel, input_channel, height, width)的顺序书写。比如，通道数为3、大小为5 × 5的滤波器有20个时，可以写成(20, 3, 5, 5)。

卷积运算中（和全连接层一样）存在偏置。在图7-11的例子中，如果进一步追加偏置的加法运算处理，则结果如下面的图7-12所示。

图7-12中，每个通道只有一个偏置。这里，偏置的形状是(*FN*, 1, 1)，滤波器的输出结果的形状是(*FN*, *OH*, *OW*)。这两个方块相加时，要对滤波器的输出结果(*FN*, *OH*, *OW*)按通道加上相同的偏置值。另外，不同形状的方块相加时，可以基于NumPy的广播功能轻松实现（1.5.5节）。

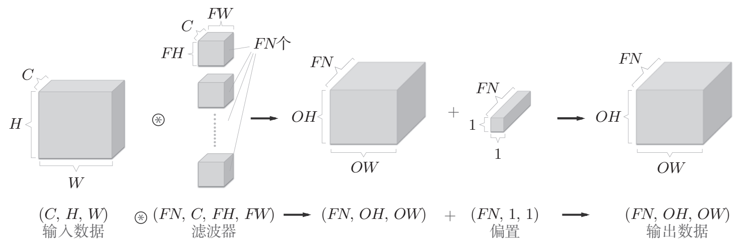


图7-12　卷积运算的处理流（追加了偏置项）

7.2.7　批处理

神经网络的处理中进行了将输入数据打包的批处理。之前的全连接神经网络的实现也对应了批处理，通过批处理，能够实现处理的高效化和学习时对mini-batch的对应。

我们希望卷积运算也同样对应批处理。为此，需要将在各层间传递的数据保存为4维数据。具体地讲，就是按(batch_num, channel, height, width)的顺序保存数据。比如，将图7-12中的处理改成对N个数据进行批处理时，数据的形状如图7-13所示。

图7-13的批处理版的数据流中，在各个数据的开头添加了批用的维度。像这样，数据作为4维的形状在各层间传递。这里需要注意的是，网络间传递的是4维数据，对这N个数据进行了卷积运算。也就是说，批处理将N次的处理汇总成了1次进行。

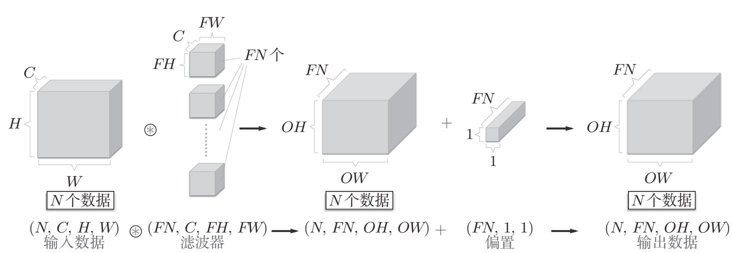


图7-13　卷积运算的处理流（批处理）

7.3　池化层

池化是缩小高、长方向上的空间的运算。比如，如图7-14所示，进行将2 × 2的区域集约成1个元素的处理，缩小空间大小。

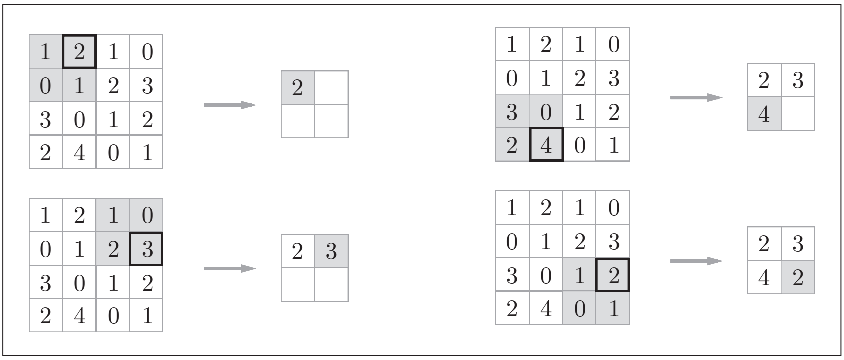


图7-14　Max池化的处理顺序

图7-14的例子是按步幅2进行2 × 2的Max池化时的处理顺序。“Max池化”是获取最大值的运算，“2 × 2”表示目标区域的大小。如图所示，从2 × 2的区域中取出最大的元素。此外，这个例子中将步幅设为了2，所以2 × 2的窗口的移动间隔为2个元素。另外，一般来说，池化的窗口大小会和步幅设定成相同的值。比如，3 × 3的窗口的步幅会设为3，4 × 4的窗口的步幅会设为4等。

除了Max池化之外，还有Average池化等。相对于Max池化是从目标区域中取出最大值，Average池化则是计算目标区域的平均值。在图像识别领域，主要使用Max池化。因此，本书中说到“池化层”时，指的是Max池化。

池化层的特征

池化层有以下特征。

没有要学习的参数

池化层和卷积层不同，没有要学习的参数。池化只是从目标区域中取最大值（或者平均值），所以不存在要学习的参数。

通道数不发生变化

经过池化运算，输入数据和输出数据的通道数不会发生变化。如图7-15所示，计算是按通道独立进行的。

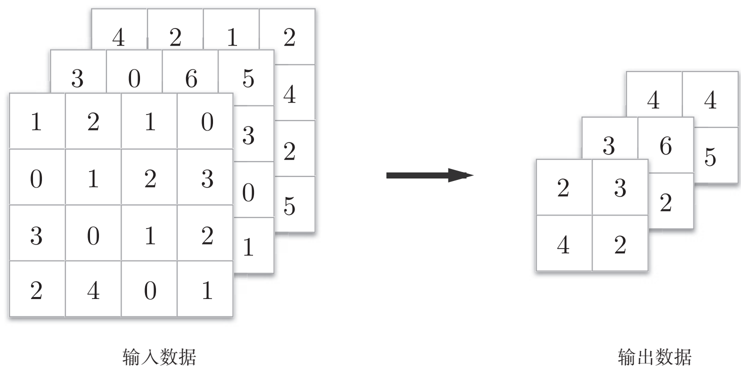


图7-15　池化中通道数不变

对微小的位置变化具有鲁棒性（健壮）

输入数据发生微小偏差时，池化仍会返回相同的结果。因此，池化对输入数据的微小偏差具有鲁棒性。比如，3×3的池化的情况下，如图7-16所示，池化会吸收输入数据的偏差（根据数据的不同，结果有可能不一致）。

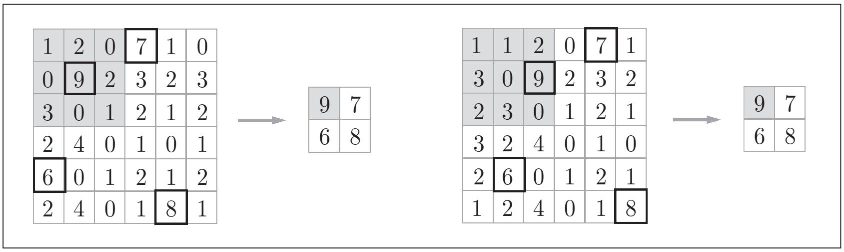


图7-16 输入数据在宽度方向上只偏离1个元素时，输出仍为相同的结果（根据数据的不同，有时结果也不相同）

7.4 卷积层和池化层的实现

前面我们详细介绍了卷积层和池化层，本节我们就用Python来实现这两个层。和第5章一样，也给进行实现的类赋予`forward`和`backward`方法，并使其可以作为模块使用。

大家可能会感觉卷积层和池化层的实现很复杂，但实际上，通过使用某种技巧，就可以很轻松地实现。本节将介绍这种技巧，将问题简化，然后再进行卷积层的实现。

7.4.1 4维数组

如前所述，CNN中各层间传递的数据是4维数据。所谓4维数据，比如数据的形状是(10, 1, 28, 28)，则它对应10个高为28、长为28、通道为1的数据。用Python来实现的话，如下所示。

```
>>> x = np.random.rand(10, 1, 28, 28) # 随机生成数据
>>> x.shape
(10, 1, 28, 28)
```

这里，如果要访问第1个数据，只要写`x[0]`就可以了（注意Python的索引是从0开始的）。同样地，用`x[1]`可以访问第2个数据。

```
>>> x[0].shape # (1, 28, 28)
>>> x[1].shape # (1, 28, 28)
```

如果要访问第1个数据的第1个通道的空间数据，可以写成下面这样。

```
>>> x[0, 0] # 或者x[0][0]
```

像这样，CNN中处理的是4维数据，因此卷积运算的实现看上去会很复杂，但是通过使用下面要介绍的`im2col`这个技巧，问题就会变得很简单。

7.4.2 基于im2col的展开

如果老老实实地实现卷积运算，估计要重复好几层的`for`语句。这样的实现有点麻烦，而且，NumPy中存在使用`for`语句后处理变慢的缺点（NumPy中，访问元素时最好不要用`for`语句）。这里，我们不使用`for`语句，而是使用`im2col`这个便利的函数进行简单的实现。

`im2col`是一个函数，将输入数据展开以适合滤波器（权重）。如图7-17所示，对3维的输入数据应用`im2col`后，数据转换为2维矩阵（正确地讲，是把包含批数量的4维数据转换成了2维数据）。

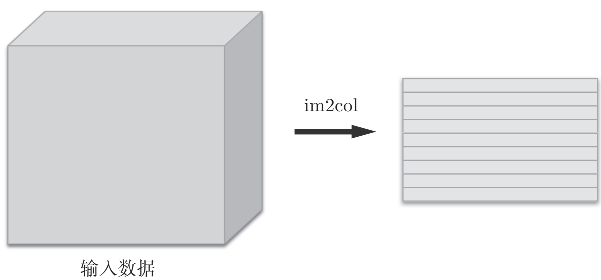


图7-17 im2col的示意图

`im2col`会把输入数据展开以适合滤波器（权重）。具体地说，如图7-18所示，对于输入数据，将应用滤波器的区域（3维方块）横向展开为1列。`im2col`会在所有应用滤波器的地方进行这个展开处理。

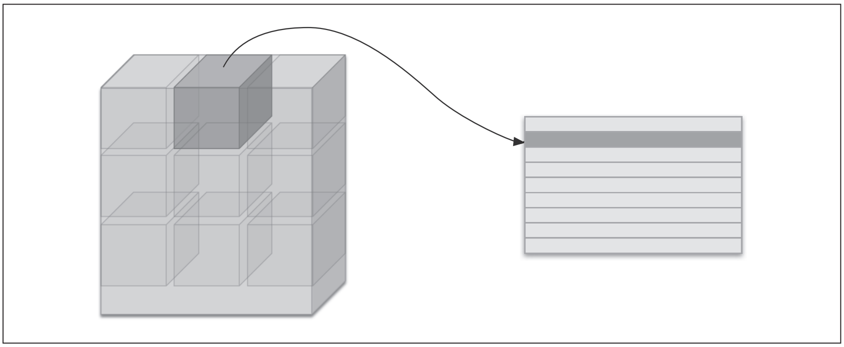


图7-18　将滤波器的应用区域从头开始依次横向展开为1列

在图7-18中，为了便于观察，将步幅设置得很大，以使滤波器的应用区域不重叠。而在实际的卷积运算中，滤波器的应用区域几乎都是重叠的。在滤波器的应用区域重叠的情况下，使用`im2col`展开后，展开后的元素个数会多于原方块的元素个数。因此，使用`im2col`的实现存在比普通的实现消耗更多内存的缺点。但是，汇总成一个大的矩阵进行计算，对计算机的计算颇有益处。比如，在矩阵计算的库（线性代数库）等中，矩阵计算的实现已被高度最优化，可以高速地进行大矩阵的乘法运算。因此，通过归结到矩阵计算上，可以有效地利用线性代数库。

> `im2col`这个名称是“image to column”的缩写，翻译过来就是“从图像到矩阵”的意思。Caffe、Chainer等深度学习框架中有名为`im2col`的函数，并且在卷积层的实现中，都使用了`im2col`。

使用`im2col`展开输入数据后，之后就只需将卷积层的滤波器（权重）纵向展开为1列，并计算2个矩阵的乘积即可（参照图7-19）。这和全连接层的Affine层进行的处理基本相同。

如图7-19所示，基于`im2col`方式的输出结果是2维矩阵。因为CNN中数据会保存为4维数组，所以要将2维输出数据转换为合适的形状。以上就是卷积层的实现流程。

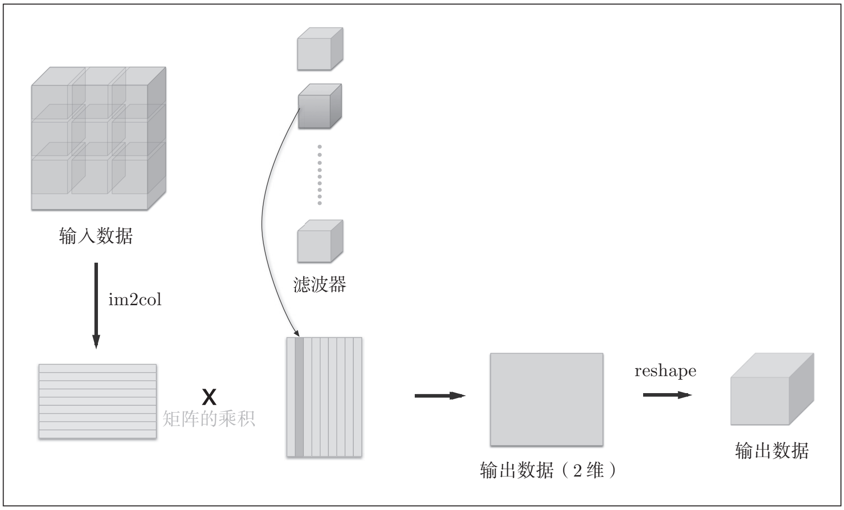


图7-19 卷积运算的滤波器处理的细节：将滤波器纵向展开为1列，并计算和im2col展开的数据的矩阵乘积，最后转换（reshape）为输出数据的大小

7.4.3 卷积层的实现

本书提供了im2col函数，并将这个im2col函数作为黑盒（不关心内部实现）使用。im2col的实现内容在common/util.py中，它的实现（实质上）是一个10行左右的简单函数。有兴趣的读者可以参考。

im2col这一便捷函数具有以下接口。

```
im2col (input_data, filter_h, filter_w, stride=1, pad=0)
```

- input_data——由（数据量，通道，高，长）的4维数组构成的输入数据
- filter_h——滤波器的高
- filter_w——滤波器的长
- stride——步幅
- pad——填充

im2col会考虑滤波器大小、步幅、填充，将输入数据展开为2维数组。现在，我们来实际使用一下这个im2col。

```
import sys, os
sys.path.append(os.pardir)
from common.util import im2col

x1 = np.random.rand(1, 3, 7, 7)
col1 = im2col(x1, 5, 5, stride=1, pad=0)
print(col1.shape) # (9, 75)

x2 = np.random.rand(10, 3, 7, 7) # 10个数据
col2 = im2col(x2, 5, 5, stride=1, pad=0)
print(col2.shape) # (90, 75)
```

这里举了两个例子。第一个是批大小为1、通道为3的7×7的数据，第二个的批大小为10，数据形状和第一个相同。分别对其应用im2col函数，在这两种情形下，第2维的元素个数均为75。这是滤波器（通道为3、大小为5×5）的元素个数的总和。批大小为1时，im2col的结果是(9, 75)。而第2个例子中批大小为10，所以保存了10倍的数据，即(90, 75)。

现在使用im2col来实现卷积层。这里我们将卷积层实现为名为Convolution的类。

```
class Convolution:
    def __init__(self, W, b, stride=1, pad=0):
        self.W = W
        self.b = b
        self.stride = stride
        self.pad = pad

    def forward(self, x):
        FN, C, FH, FW = self.W.shape
        N, C, H, W = x.shape
        out_h = int(1 + (H + 2*self.pad - FH) / self.stride)
        out_w = int(1 + (W + 2*self.pad - FW) / self.stride)

        col = im2col(x, FH, FW, self.stride, self.pad)
        col_W = self.W.reshape(FN, -1).T # 滤波器的展开
        out = np.dot(col, col_W) + self.b

        out = out.reshape(N, out_h, out_w, -1).transpose(0, 3, 1, 2)

        return out
```

卷积层的初始化方法将滤波器(权重)、偏置、步幅、填充作为参数接收。滤波器是 `(FN, C, FH, FW)` 的4维形状。另外，`FN`、`C`、`FH`、`FW` 分别是 Filter Number(滤波器数量)、Channel、Filter Height、Filter Width 的缩写。

这里用粗体字表示 Convolution 层的实现中的重要部分。在这些粗体字部分，用 `im2col` 展开输入数据，并用 `reshape` 将滤波器展开为2维数组。然后，计算展开后的矩阵的乘积。

展开滤波器的部分(代码段中的粗体字)如图 7-19 所示，将各个滤波器的方块纵向展开为1列。这里通过 `reshape(FN,-1)` 将参数指定为 `-1`，这是 `reshape` 的一个便利的功能。通过在 `reshape` 时指定为 `-1`，`reshape` 函数会自动计算 `-1` 维度上的元素个数，以使多维数组的元素个数前后一致。比如，(10, 3, 5, 5) 形状的数组的元素个数共有 750 个，指定 `reshape(10,-1)` 后，就会转换成 (10, 75) 形状的数组。

`forward` 的实现中，最后会将输出大小转换为合适的形状。转换时使用了 NumPy 的 `transpose` 函数。`transpose` 会更改多维数组的轴的顺序。如图 7-20 所示，通过指定从0开始的索引(编号)序列，就可以更改轴的顺序。

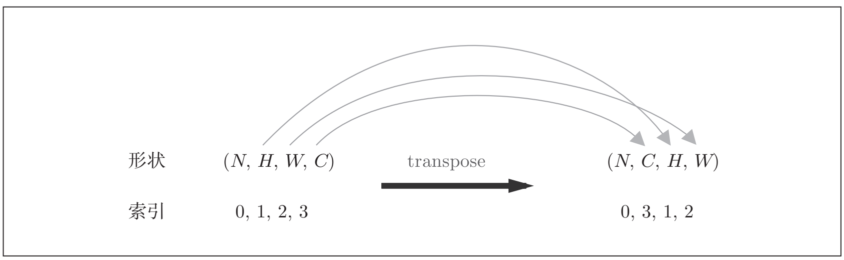


图 7-20 基于 NumPy 的 transpose 的轴顺序的更改：通过指定索引(编号)，更改轴的顺序

以上就是卷积层的 `forward` 处理的实现。通过使用 `im2col` 进行展开，基本上可以像实现全连接层的 Affine 层一样来实现(5.6 节)。接下来是卷积层的反向传播的实现，因为和 Affine 层的实现有很多共通的地方，所以就不再介绍了。但有一点需要注意，在进行卷积层的反向传播时，必须进行 `im2col` 的逆处理。这可以使用本书提供的 `col2im` 函数(`col2im` 的实现在 `common/util.`

py中）来进行。除了使用col2im这一点，卷积层的反向传播和Affine层的实现方式都一样。卷积层的反向传播的实现在common/layer.py中，有兴趣的读者可以参考。

7.4.4 池化层的实现

池化层的实现和卷积层相同，都使用im2col展开输入数据。不过，池化的情况下，在通道方向上是独立的，这一点和卷积层不同。具体地讲，如图7-21所示，池化的应用区域按通道单独展开。

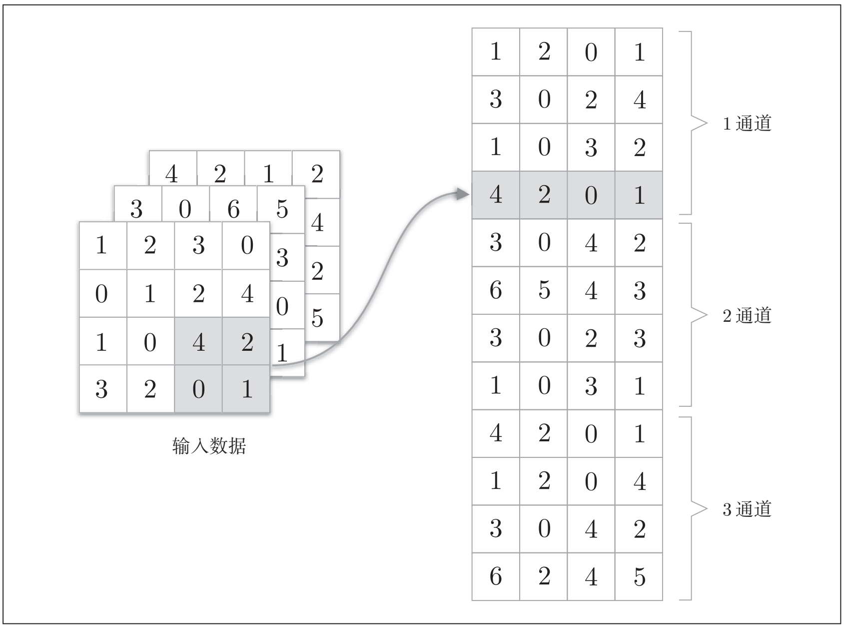


图7-21 对输入数据展开池化的应用区域（2×2的池化的例子）

像这样展开之后，只需对展开的矩阵求各行的最大值，并转换为合适的形状即可（图7-22）。

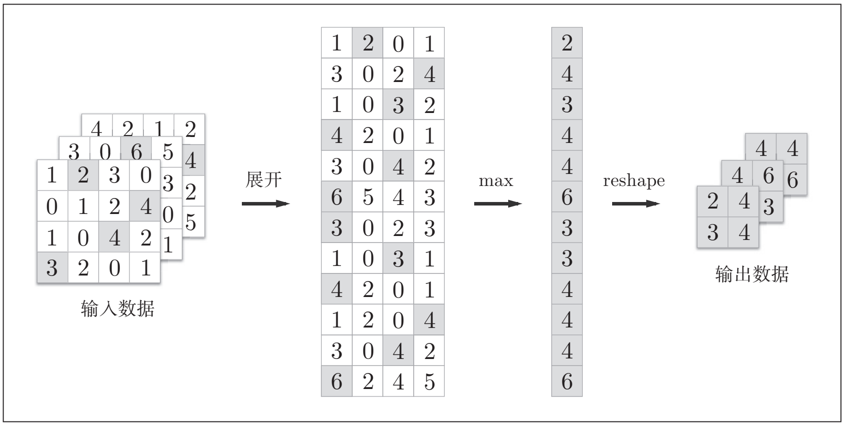


图7-22　池化层的实现流程：池化的应用区域内的最大值元素用灰色表示

上面就是池化层的forward处理的实现流程。下面来看一下Python的实现示例。

```
class Pooling:
    def __init__(self, pool_h, pool_w, stride=1, pad=0):
        self.pool_h = pool_h
        self.pool_w = pool_w
        self.stride = stride
        self.pad = pad

    def forward(self, x):
        N, C, H, W = x.shape
        out_h = int(1 + (H - self.pool_h) / self.stride)
        out_w = int(1 + (W - self.pool_w) / self.stride)

        # 展开(1)
        col = im2col(x, self.pool_h, self.pool_w, self.stride, self.pad)
        col = col.reshape(-1, self.pool_h*self.pool_w)

        # 最大值(2)
        out = np.max(col, axis=1)
        # 转换(3)
        out = out.reshape(N, out_h, out_w, C).transpose(0, 3, 1, 2)

        return out
```

如图7-22所示，池化层的实现按下面3个阶段进行。

1. 展开输入数据。

2. 求各行的最大值。

3. 转换为合适的输出大小。

各阶段的实现都很简单，只有一两行代码。

> 最大值的计算可以使用NumPy的`np.max`方法。`np.max`可以指定`axis`参数，并在这个参数指定的各个轴方向上求最大值。比如，如果写成`np.max(x, axis=1)`，就可以在输入x的第1维的各个轴方向上求最大值。

以上就是池化层的`forward`处理的介绍。如上所述，通过将输入数据展开为容易进行池化的形状，后面的实现就会变得非常简单。

关于池化层的`backward`处理，之前已经介绍过相关内容，这里就不再介绍了。另外，池化层的`backward`处理可以参考ReLU层的实现中使用的max的反向传播（5.5.1节）。池化层的实现在`common/layer.py`中，有兴趣的读者可以参考。

7.5 CNN的实现

我们已经实现了卷积层和池化层，现在来组合这些层，搭建进行手写数字识别的CNN。这里要实现如图7-23所示的CNN。

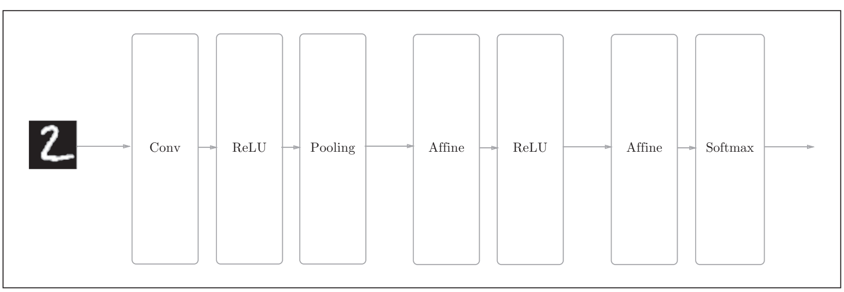


图7-23 简单CNN的网络构成

如图7-23所示，网络的构成是“Convolution - ReLU - Pooling -Affine - ReLU - Affine - Softmax”，我们将它实现为名为SimpleConvNet的类。

首先来看一下SimpleConvNet的初始化（__init__），取下面这些参数。

参数

- input_dim——输入数据的维度：（通道，高，长）
- conv_param——卷积层的超参数（字典）。字典的关键字如下：
  filter_num——滤波器的数量
  filter_size——滤波器的大小
  stride——步幅
  pad——填充
- hidden_size——隐藏层（全连接）的神经元数量
- output_size——输出层（全连接）的神经元数量
- weitght_int_std——初始化时权重的标准差

这里，卷积层的超参数通过名为conv_param的字典传入。我们设想它会像{'filter_num':30,'filter_size':5, 'pad':0, 'stride':1}这样，保存必要的超参数值。

SimpleConvNet的初始化的实现稍长，我们分成3部分来说明，首先是初始化的最开始部分。

```
class SimpleConvNet:
    def __init__(self, input_dim=(1, 28, 28),
                 conv_param={'filter_num':30, 'filter_size':5,
                             'pad':0, 'stride':1},
                 hidden_size=100, output_size=10, weight_init_std=0.01):
        filter_num = conv_param['filter_num']
        filter_size = conv_param['filter_size']
        filter_pad = conv_param['pad']
        filter_stride = conv_param['stride']
        input_size = input_dim[1]
        conv_output_size = (input_size - filter_size + 2*filter_pad) / \
                            filter_stride + 1
        pool_output_size = int(filter_num * (conv_output_size/2) *
                               (conv_output_size/2))
```

这里将由初始化参数传入的卷积层的超参数从字典中取了出来（以方便后面使用），然后，计算卷积层的输出大小。接下来是权重参数的初始化部分。

```
self.params = {}
self.params['W1'] = weight_init_std * \
                    np.random.randn(filter_num, input_dim[0],
                                    filter_size, filter_size)
self.params['b1'] = np.zeros(filter_num)
self.params['W2'] = weight_init_std * \
                    np.random.randn(pool_output_size,
                                    hidden_size)
self.params['b2'] = np.zeros(hidden_size)
self.params['W3'] = weight_init_std * \
                    np.random.randn(hidden_size, output_size)
self.params['b3'] = np.zeros(output_size)
```

学习所需的参数是第1层的卷积层和剩余两个全连接层的权重和偏置。将这些参数保存在实例变量的params字典中。将第1层的卷积层的权重设为关键字W1，偏置设为关键字b1。同样，分别用关键字W2、b2和关键字W3、b3来保存第2个和第3个全连接层的权重和偏置。

最后，生成必要的层。

```
self.layers = OrderedDict()
self.layers['Conv1'] = Convolution(self.params['W1'],
                                   self.params['b1'],
                                   conv_param['stride'],
                                   conv_param['pad'])

self.layers['Relu1'] = Relu()
self.layers['Pool1'] = Pooling(pool_h=2, pool_w=2, stride=2)
self.layers['Affine1'] = Affine(self.params['W2'],
                              self.params['b2'])

self.layers['Relu2'] = Relu()
self.layers['Affine2'] = Affine(self.params['W3'],
                              self.params['b3'])
self.last_layer = softmaxwithloss()
```

从最前面开始按顺序向有序字典（OrderedDict）的layers中添加层。只有最后的SoftmaxWithLoss层被添加到别的变量lastLayer中。

以上就是SimpleConvNet的初始化中进行的处理。像这样初始化后，进行推理的predict方法和求损失函数值的loss方法就可以像下面这样实现。

```
def predict(self, x):
    for layer in self.layers.values():
        x = layer.forward(x)
    return x

def loss(self, x, t):
    y = self.predict(x)
    return self.lastLayer.forward(y, t)
```

这里，参数x是输入数据，t是教师标签。用于推理的predict方法从头开始依次调用已添加的层，并将结果传递给下一层。在求损失函数的loss方法中，除了使用predict方法进行的forward处理之外，还会继续进行forward处理，直到到达最后的SoftmaxWithLoss层。

接下来是基于误差反向传播法求梯度的代码实现。

```
def gradient(self, x, t):
    # forward
    self.loss(x, t)

    # backward
    dout = 1
    dout = self.lastLayer.backward(dout)

    layers = list(self.layers.values())
    layers.reverse()
    for layer in layers:
        dout = layer.backward(dout)

    # 设定
    grads = {}
    grads['W1'] = self.layers['Conv1'].dW
    grads['b1'] = self.layers['Conv1'].db
    grads['W2'] = self.layers['Affine1'].dW
    grads['b2'] = self.layers['Affine1'].db
    grads['W3'] = self.layers['Affine2'].dW
    grads['b3'] = self.layers['Affine2'].db

    return grads
```

参数的梯度通过误差反向传播法(反向传播)求出，通过把正向传播和反向传播组装在一起来完成。因为已经在各层正确实现了正向传播和反向传播的功能，所以这里只需要以合适的顺序调用即可。最后，把各个权重参数的梯度保存到grads字典中。这就是SimpleConvNet的实现。

现在，使用这个`SimpleConvNet`学习MNIST数据集。用于学习的代码与4.5节中介绍的代码基本相同，因此这里不再罗列（源代码在`ch07/train_convnet.py`中）。

如果使用MNIST数据集训练`SimpleConvNet`，则训练数据的识别率为99.82%，测试数据的识别率为98.96%（每次学习的识别精度都会发生一些误差）。测试数据的识别率大约为99%，就小型网络来说，这是一个非常高的识别率。下一章，我们会通过进一步叠加层来加深网络，实现测试数据的识别率超过99%的网络。

如上所述，卷积层和池化层是图像识别中必备的模块。CNN可以有效读取图像中的某种特性，在手写数字识别中，还可以实现高精度的识别。

7.6 CNN的可视化

CNN中用到的卷积层在"观察"什么呢？本节将通过卷积层的可视化，探索CNN中到底进行了什么处理。

7.6.1 第1层权重的可视化

刚才我们对MNIST数据集进行了简单的CNN学习。当时，第1层的卷积层的权重的形状是$(30, 1, 5, 5)$，即30个大小为5×5、通道为1的滤波器。滤波器大小是5×5、通道数是1，意味着滤波器可以可视化为1通道的灰度图像。现在，我们将卷积层（第1层）的滤波器显示为图像。这里，我们来比较一下学习前和学习后的权重，结果如图7-24所示（源代码在`ch07/visualize_filter.py`中）。

图7-24中，学习前的滤波器是随机进行初始化的，所以在黑白的浓淡上没有规律可循，但学习后的滤波器变成了有规律的图像。我们发现，通过学习，滤波器被更新成了有规律的滤波器，比如从白到黑渐变的滤波器、含有块状区域（称为blob）的滤波器等。

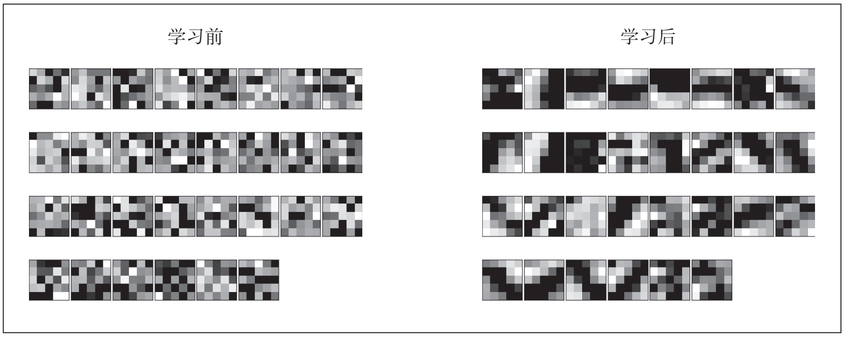


图7-24 学习前和学习后的第1层的卷积层的权重：虽然权重的元素是实数，但是在图像的显示上，统一将最小值显示为黑色(0)，最大值显示为白色(255)

如果要问图7-24中右边的有规律的滤波器在“观察”什么，答案就是它在观察边缘(颜色变化的分界线)和斑块(局部的块状区域)等。比如，左半部分为白色、右半部分为黑色的滤波器的情况下，如图7-25所示，会对垂直方向上的边缘有响应。

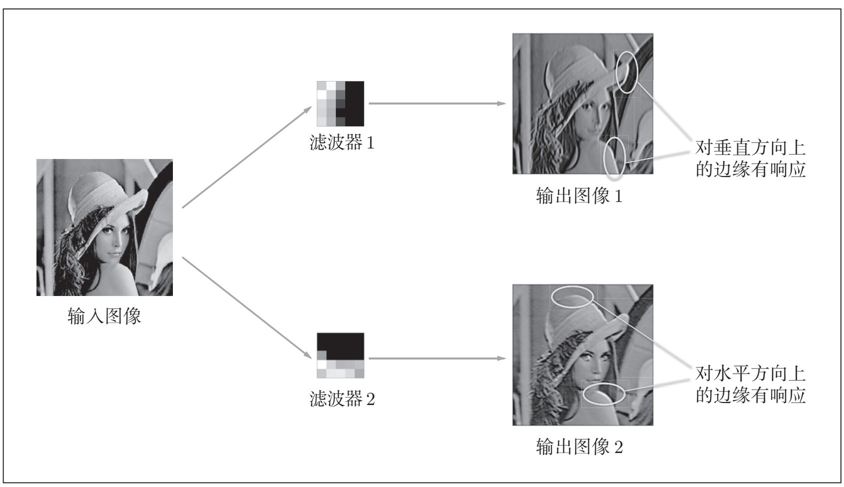


图7-25 对水平方向上和垂直方向上的边缘有响应的滤波器：输出图像1中，垂直方向的边缘上出现白色像素，输出图像2中，水平方向的边缘上出现很多白色像素

图7-25中显示了选择两个学习完的滤波器对输入图像进行卷积处理时的结果。我们发现“滤波器1”对垂直方向上的边缘有响应，“滤波器2”对水平方向上的边缘有响应。

由此可知，卷积层的滤波器会提取边缘或斑块等原始信息。而刚才实现的CNN会将这些原始信息传递给后面的层。

7.6.2　基于分层结构的信息提取

上面的结果是针对第1层的卷积层得出的。第1层的卷积层中提取了边缘或斑块等“低级”信息，那么在堆叠了多层的CNN中，各层中又会提取什么样的信息呢？根据深度学习的可视化相关的研究[17][18]，随着层次加深，提取的信息（正确地讲，是反映强烈的神经元）也越来越抽象。

图7-26中展示了进行一般物体识别（车或狗等）的8层CNN。这个网络结构的名称是下一节要介绍的AlexNet。AlexNet网络结构堆叠了多层卷积层和池化层，最后经过全连接层输出结果。图7-26的方块表示的是中间数据，对于这些中间数据，会连续应用卷积运算。

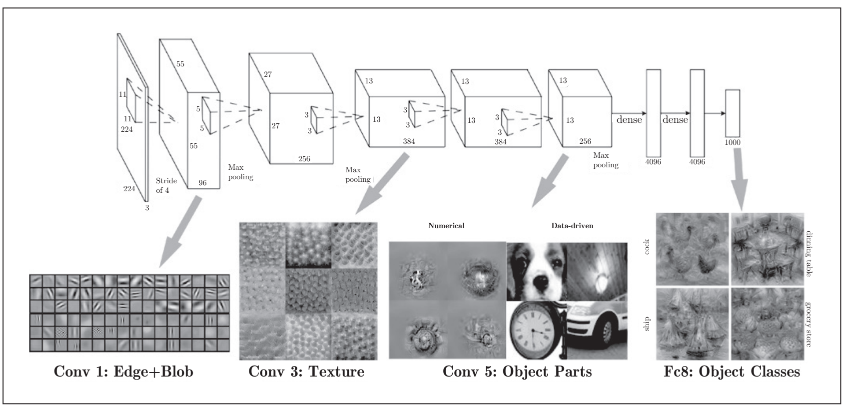


图7-26　CNN的卷积层中提取的信息。第1层的神经元对边缘或斑块有响应，第3层对纹理有响应，第5层对物体部件有响应，最后的全连接层对物体的类别（狗或车）有响应（图像引用自文献[19]）

如图7-26所示，如果堆叠了多层卷积层，则随着层次加深，提取的信息也愈加复杂、抽象，这是深度学习中很有意思的一个地方。最开始的层对简单的边缘有响应，接下来的层对纹理有响应，再后面的层对更加复杂的物体部件有响应。也就是说，随着层次加深，神经元从简单的形状向"高级"信息变化。换句话说，就像我们理解东西的"含义"一样，响应的对象在逐渐变化。

7.7 具有代表性的CNN

关于CNN，迄今为止已经提出了各种网络结构。这里，我们介绍其中特别重要的两个网络，一个是在1998年首次被提出的CNN元祖LeNet[20]，另一个是在深度学习受到关注的2012年被提出的AlexNet[21]。

7.7.1 LeNet

LeNet在1998年被提出，是进行手写数字识别的网络。如图7-27所示，它有连续的卷积层和池化层（正确地讲，是只"抽选元素"的子采样层），最后经全连接层输出结果。

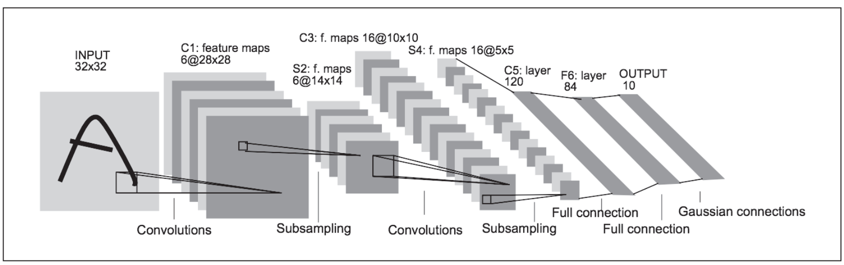


图7-27 LeNet的网络结构（引用自文献[20]）

和"现在的CNN"相比，LeNet有几个不同点。第一个不同点在于激活函数。LeNet中使用sigmoid函数，而现在的CNN中主要使用ReLU函数。此外，原始的LeNet中使用子采样（subsampling）缩小中间数据的大小，而现在的CNN中Max池化是主流。

综上，LeNet与现在的CNN虽然有些许不同，但差别并不是那么大。想到LeNet是20多年前提出的最早的CNN，还是很令人称奇的。

7.7.2 AlexNet

在LeNet问世20多年后，AlexNet被发布出来。AlexNet是引发深度学习热潮的导火线，不过它的网络结构和LeNet基本上没有什么不同，如图7-28所示。

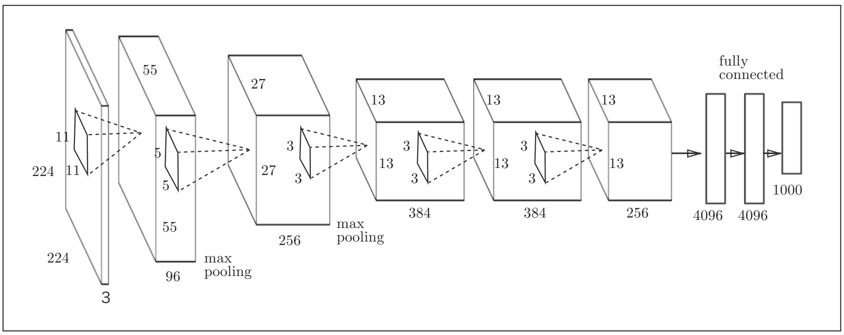


图7-28 AlexNet（根据文献[21]生成）

AlexNet叠有多个卷积层和池化层，最后经由全连接层输出结果。虽然结构上AlexNet和LeNet没有大的不同，但有以下几点差异。

- 激活函数使用ReLU。
- 使用进行局部正规化的LRN（Local Response Normalization）层。
- 使用Dropout（6.4.3节）。

如上所述，关于网络结构，LeNet和AlexNet没有太大的不同。但是，围绕它们的环境和计算机技术有了很大的进步。具体地说，现在任何人都可以获得大量的数据。而且，擅长大规模并行计算的GPU得到普及，高速进行大量的运算已经成为可能。大数据和GPU已成为深度学习发展的巨大的原动力。

大多数情况下，深度学习（加深了层次的网络）存在大量的参数。因此，学习需要大量的计算，并且需要使那些参数“满意”的大量数据。可以说是GPU和大数据给这些课题带来了希望。

7.8 小结

本章介绍了CNN。构成CNN的基本模块的卷积层和池化层虽然有些复杂，但是一旦理解了，之后就只是如何使用它们的问题了。本章为了使读者在实现层面上理解卷积层和池化层，花了不少时间进行介绍。在图像处理领域，几乎毫无例外地都会使用CNN。请扎实地理解本章的内容，然后进入最后一章的学习。

本章所学的内容

- CNN在此前的全连接层的网络中新增了卷积层和池化层。
- 使用`im2col`函数可以简单、高效地实现卷积层和池化层。
- 通过CNN的可视化，可知随着层次变深，提取的信息愈加高级。
- LeNet和AlexNet是CNN的代表性网络。
- 在深度学习的发展中，大数据和GPU做出了很大的贡献。

第 8 章
深度学习

深度学习是加深了层的深度神经网络。基于之前介绍的网络，只需通过叠加层，就可以创建深度网络。本章我们将看一下深度学习的性质、课题和可能性，然后对当前的深度学习进行概括性的说明。

8.1 加深网络

关于神经网络，我们已经学了很多东西，比如构成神经网络的各种层、学习时的有效技巧、对图像特别有效的CNN、参数的最优化方法等，这些都是深度学习中的重要技术。本节我们将这些已经学过的技术汇总起来，创建一个深度网络，挑战MNIST数据集的手写数字识别。

8.1.1 向更深的网络出发

话不多说，这里我们来创建一个如图8-1所示的网络结构的CNN（一个比之前的网络都深的网络）。这个网络参考了下一节要介绍的VGG。

如图8-1所示，这个网络的层比之前实现的网络都更深。这里使用的卷积层全都是3×3的小型滤波器，特点是随着层的加深，通道数变大（卷积层的通道数从前面的层开始按顺序以16、16、32、32、64、64的方式增加）。此外，如图8-1所示，插入了池化层，以逐渐减小中间数据的空间大小；并且，后面的全连接层中使用了Dropout层。

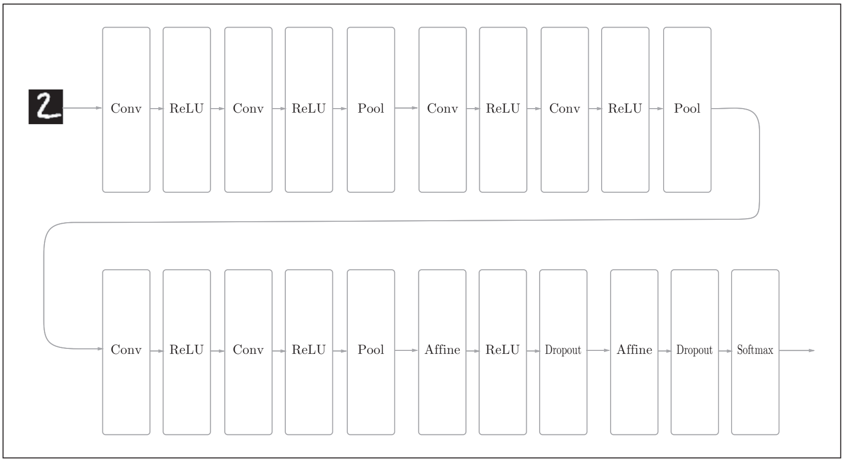


图8-1 进行手写数字识别的深度CNN

这个网络使用He初始值作为权重的初始值，使用Adam更新权重参数。把上述内容总结起来，这个网络有如下特点。

- 基于3×3的小型滤波器的卷积层。
- 激活函数是ReLU。
- 全连接层的后面使用Dropout层。
- 基于Adam的最优化。
- 使用He初始值作为权重初始值。

从这些特征中可以看出，图8-1的网络中使用了多个之前介绍的神经网络技术。现在，我们使用这个网络进行学习。先说一下结论，这个网络的识别精度为99.38% ①，可以说是非常优秀的性能了！

① 最终的识别精度有少许偏差，不过在这个网络中，识别精度大体上都会超过99%。

实现图8-1的网络的源代码在 `ch08/deep_convnet.py` 中，训练用的代码在 `ch08/train_deepnet.py` 中。虽然使用这些代码可以重现这里进行的学习，不过深度网络的学习需要花费较多的时间（大概要半天以上）。本书以 `ch08/deep_conv_net_params.pkl` 的形式给出了学习完的权重参数。刚才的 `deep_convnet.py` 备有读入学习完的参数的功能，请根据需要进行使用。

图8-1的网络的错误识别率只有0.62%。这里我们实际看一下在什么样的图像上发生了识别错误。图8-2中显示了识别错误的例子。

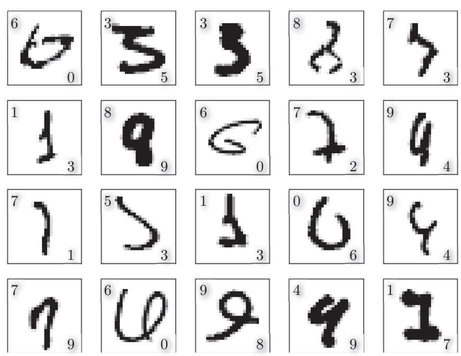


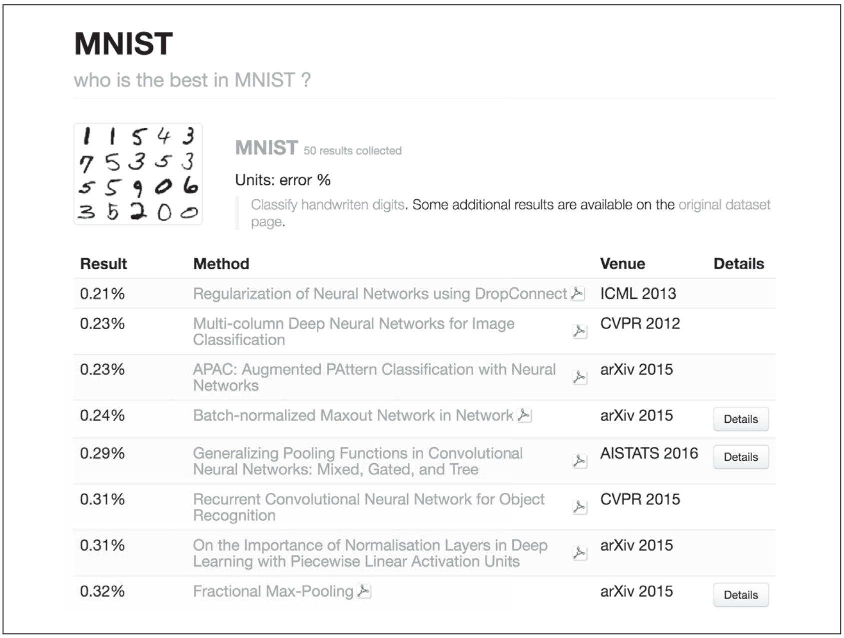
图8-2　识别错误的图像的例子：各个图像的左上角显示了正确解标签，右下角显示了本网络的推理结果

观察图8-2可知，这些图像对于我们人类而言也很难判断。实际上，这里面有几个图像很难判断是哪个数字，即使是我们人类，也同样会犯“识别错误”。比如，左上角的图像（正确解是“6”）看上去像“0”，它旁边的图像（正确解是“3”）看上去像“5”。整体上，“1”和“7”、“0”和“6”、“3”和“5”的组合比较容易混淆。通过这些例子，相信大家可以理解为何会发生识别错误了吧。

这次的深度CNN尽管识别精度很高，但是对于某些图像，也犯了和人类同样的“识别错误”。从这一点上，我们也可以感受到深度CNN中蕴藏着巨大的可能性。

8.1.2 进一步提高识别精度

在一个标题为“What is the class of this image ?”的网站[32]上，以排行榜的形式刊登了目前为止通过论文等渠道发表的针对各种数据集的方法的识别精度（图8-3）。



MNIST

who is the best in MNIST ?

MNIST 50 results collected

Units: error %

Classify handwriten digits. Some additional results are available on the original dataset page.

Result	Method	Venue	Details
0.21%	Regularization of Neural Networks using DropConnect	ICML 2013	
0.23%	Multi-column Deep Neural Networks for Image Classification	CVPR 2012	
0.23%	APAC: Augmented PAttern Classification with Neural Networks	arXiv 2015	
0.24%	Batch-normalized Maxout Network in Network	arXiv 2015	Details
0.29%	Generalizing Pooling Functions in Convolutional Neural Networks: Mixed, Gated, and Tree	AISTATS 2016	Details
0.31%	Recurrent Convolutional Neural Network for Object Recognition	CVPR 2015	
0.31%	On the Importance of Normalisation Layers in Deep Learning with Piecewise Linear Activation Units	arXiv 2015	
0.32%	Fractional Max-Pooling	arXiv 2015	Details

图8-3 针对MNIST数据集的各种方法的排行（引自文献[32]：2016年6月）

观察图8-3的排行结果，可以发现“Neural Networks”“Deep”“Convolutional”等关键词特别显眼。实际上，排行榜上的前几名大都是基于CNN的方法。顺便说一下，截止到2016年6月，对MNIST数据集的最高识别精度是99.79%（错误识别率为0.21%），该方法也是以CNN为基础的[33]。不过，它用的CNN并不是特别深层的网络（卷积层为2层、全连接层为2层的网络）。

对于MNIST数据集，层不用特别深就获得了（目前）最高的识别精度。一般认为，这是因为对于手写数字识别这样一个比较简单的任务，没有必要将网络的表现力提高到那么高的程度。因此，可以说加深层的好处并不大。而之后要介绍的大规模的一般物体识别的情况，因为问题复杂，所以加深层对提高识别精度大有裨益。

参考刚才排行榜中前几名的方法，可以发现进一步提高识别精度的技术和线索。比如，集成学习、学习率衰减、**Data Augmentation**（数据扩充）等都有助于提高识别精度。尤其是Data Augmentation，虽然方法很简单，但在提高识别精度上效果显著。

Data Augmentation基于算法“人为地”扩充输入图像（训练图像）。具体地说，如图8-4所示，对于输入图像，通过施加旋转、垂直或水平方向上的移动等微小变化，增加图像的数量。这在数据集的图像数量有限时尤其有效。

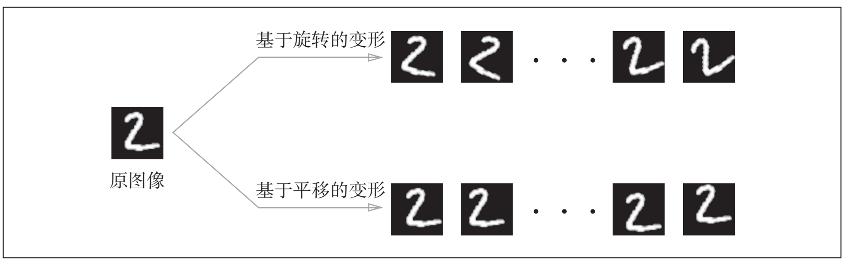


图8-4　Data Augmentation的例子

除了如图8-4所示的变形之外，Data Augmentation还可以通过其他各种方法扩充图像，比如裁剪图像的“crop处理”、将图像左右翻转的“flip处理”[①]等。对于一般的图像，施加亮度等外观上的变化、放大缩小等尺度上的变化也是有效的。不管怎样，通过Data Augmentation巧妙地增加训练图像，就可以提高深度学习的识别精度。虽然这个看上去只是一个简单的技巧，不过经常会有很好的效果。这里，我们不进行Data Augmentation的实现，不

① flip处理只在不需要考虑图像对称性的情况下有效。

过这个技巧的实现比较简单，有兴趣的读者请自己试一下。

8.1.3 加深层的动机

关于加深层的重要性，现状是理论研究还不够透彻。尽管目前相关理论还比较贫乏，但是有几点可以从过往的研究和实验中得以解释（虽然有一些直观）。本节就加深层的重要性，给出一些增补性的数据和说明。

首先，从以ILSVRC为代表的大规模图像识别的比赛结果中可以看出加深层的重要性（详细内容请参考下一节）。这种比赛的结果显示，最近前几名的方法多是基于深度学习的，并且有逐渐加深网络的层的趋势。也就是说，可以看到层越深，识别性能也越高。

下面我们说一下加深层的好处。其中一个好处就是可以减少网络的参数数量。说得详细一点，就是与没有加深层的网络相比，加深了层的网络可以用更少的参数达到同等水平（或者更强）的表现力。这一点结合卷积运算中的滤波器大小来思考就好理解了。比如，图8-5展示了由 5×5 的滤波器构成的卷积层。

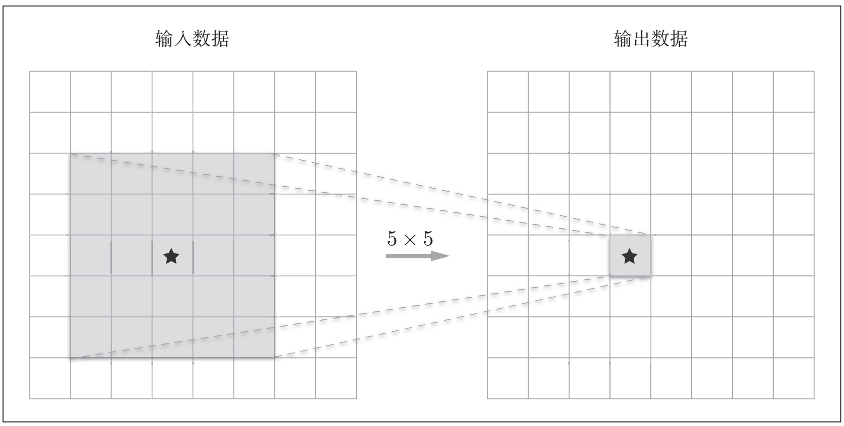


图8-5 5×5的卷积运算的例子

这里希望大家考虑一下输出数据的各个节点是从输入数据的哪个区域计算出来的。显然，在图8-5的例子中，每个输出节点都是从输入数据的某个5 × 5的区域算出来的。接下来我们思考一下图8-6中重复两次3 × 3的卷积运算的情形。此时，每个输出节点将由中间数据的某个3 × 3的区域计算出来。那么，中间数据的3 × 3的区域又是由前一个输入数据的哪个区域计算出来的呢？仔细观察图8-6，可知它对应一个5 × 5的区域。也就是说，图8-6的输出数据是“观察”了输入数据的某个5 × 5的区域后计算出来的。

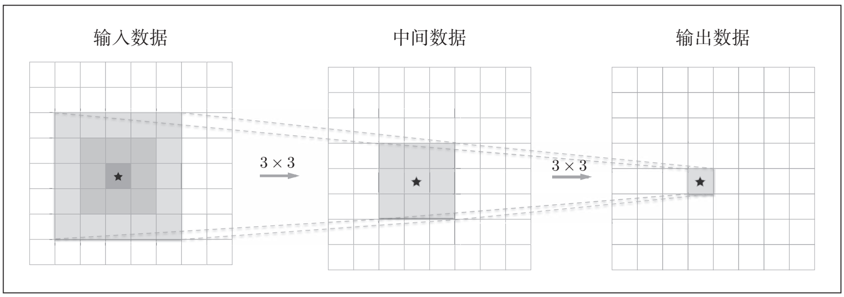


图8-6　重复两次3×3的卷积层的例子

一次5 × 5的卷积运算的区域可以由两次3 × 3的卷积运算抵充。并且，相对于前者的参数数量25（5 × 5），后者一共是18（2 × 3 × 3），通过叠加卷积层，参数数量减少了。而且，这个参数数量之差会随着层的加深而变大。比如，重复三次3 × 3的卷积运算时，参数的数量总共是27。而为了用一次卷积运算“观察”与之相同的区域，需要一个7 × 7的滤波器，此时的参数数量是49。



叠加小型滤波器来加深网络的好处是可以减少参数的数量，扩大**感受野**（receptive field，给神经元施加变化的某个局部空间区域）。并且，通过叠加层，将ReLU等激活函数夹在卷积层的中间，进一步提高了网络的表现力。这是因为向网络添加了基于激活函数的“非线性”表现力，通过非线性函数的叠加，可以表现更加复杂的东西。

加深层的另一个好处就是使学习更加高效。与没有加深层的网络相比，通过加深层，可以减少学习数据，从而高效地进行学习。为了直观地理解这一点，大家可以回忆一下7.6节的内容。7.6节中介绍了CNN的卷积层会分层次地提取信息。具体地说，在前面的卷积层中，神经元会对边缘等简单的形状有响应，随着层的加深，开始对纹理、物体部件等更加复杂的东西有响应。

我们先牢记这个网络的分层结构，然后考虑一下"狗"的识别问题。要用浅层网络解决这个问题的话，卷积层需要一下子理解很多"狗"的特征。"狗"有各种各样的种类，根据拍摄环境的不同，外观变化也很大。因此，要理解"狗"的特征，需要大量富有差异性的学习数据，而这会导致学习需要花费很多时间。

不过，通过加深网络，就可以分层次地分解需要学习的问题。因此，各层需要学习的问题就变成了更简单的问题。比如，最开始的层只要专注于学习边缘就好，这样一来，只需用较少的学习数据就可以高效地进行学习。这是为什么呢？因为和印有"狗"的照片相比，包含边缘的图像数量众多，并且边缘的模式比"狗"的模式结构更简单。

通过加深层，可以分层次地传递信息，这一点也很重要。比如，因为提取了边缘的层的下一层能够使用边缘的信息，所以应该能够高效地学习更加高级的模式。也就是说，通过加深层，可以将各层要学习的问题分解成容易解决的简单问题，从而可以进行高效的学习。

以上就是对加深层的重要性的增补性说明。不过，这里需要注意的是，近几年的深层化是由大数据、计算能力等即便加深层也能正确地进行学习的新技术和环境支撑的。

8.2 深度学习的小历史

一般认为，现在深度学习之所以受到大量关注，其契机是2012年举办的大规模图像识别大赛ILSVRC（ImageNet Large Scale Visual Recognition Challenge）。在那年的比赛中，基于深度学习的方法（通称AlexNet）以压倒性的优势胜出，彻底颠覆了以往的图像识别方法。2012年深度学习的这场逆

袭成为一个转折点，在之后的比赛中，深度学习一直活跃在舞台中央。本节我们以ILSVRC这个大规模图像识别比赛为轴，看一下深度学习最近的发展趋势。

8.2.1 ImageNet

ImageNet[25]是拥有超过100万张图像的数据集。如图8-7所示，它包含了各种各样的图像，并且每张图像都被关联了标签（类别名）。每年都会举办使用这个巨大数据集的ILSVRC图像识别大赛。

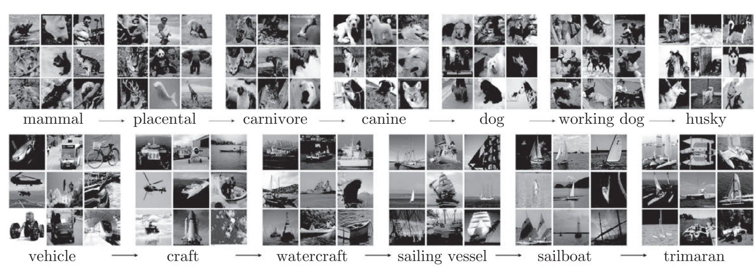


图8-7　大规模数据集ImageNet的数据例（引用自文献[25]）

ILSVRC大赛有多个测试项目，其中之一是“类别分类”（classification），在该项目中，会进行1000个类别的分类，比试识别精度。我们来看一下最近几年的ILSVRC大赛的类别分类项目的结果。图8-8中展示了从2010年到2015年的优胜队伍的成绩。这里，将前5类中出现正确解的情况视为“正确”，此时的错误识别率用柱形图来表示。

图8-8中需要注意的是，以2012年为界，之后基于深度学习的方法一直居于首位。实际上，我们发现2012年的AlexNet大幅降低了错误识别率。并且，此后基于深度学习的方法不断在提升识别精度。特别是2015年的ResNet（一个超过150层的深度网络）将错误识别率降低到了3.5%。据说这个结果甚至超过了普通人的识别能力。

这些年深度学习取得了不斐的成绩，其中VGG、GoogLeNet、ResNet

已广为人知，在与深度学习有关的各种场合都会遇到这些网络。下面我们就来简单地介绍一下这3个有名的网络。

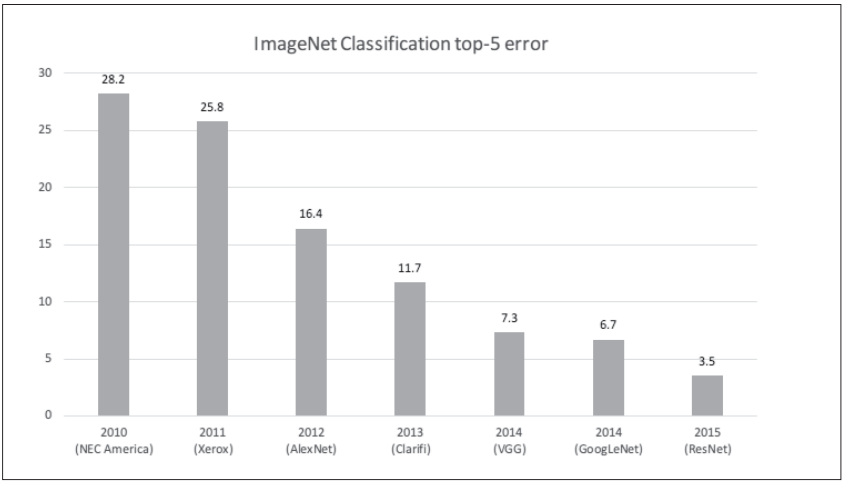


图8-8　ILSCRV优胜队伍的成绩演变：竖轴是错误识别率，横轴是年份。横轴的括号内是队伍名或者方法名

8.2.2　VGG

VGG是由卷积层和池化层构成的基础的CNN。不过，如图8-9所示，它的特点在于将有权重的层（卷积层或者全连接层）叠加至16层（或者19层），具备了深度（根据层的深度，有时也称为“VGG16”或“VGG19”）。

VGG中需要注意的地方是，基于3×3的小型滤波器的卷积层的运算是连续进行的。如图8-9所示，重复进行“卷积层重叠2次到4次，再通过池化层将大小减半”的处理，最后经由全连接层输出结果。

VGG在2014年的比赛中最终获得了第2名的成绩（下一节介绍的GoogleNet是2014年的第1名）。虽然在性能上不及GoogleNet，但因为VGG结构简单，应用性强，所以很多技术人员都喜欢使用基于VGG的网络。

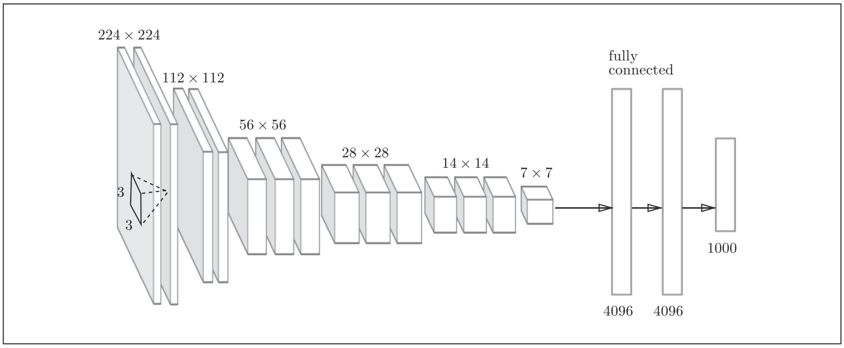


图8-9　VGG（根据文献[22]生成）

8.2.3 GoogLeNet

GoogLeNet的网络结构如图8-10所示。图中的矩形表示卷积层、池化层等。

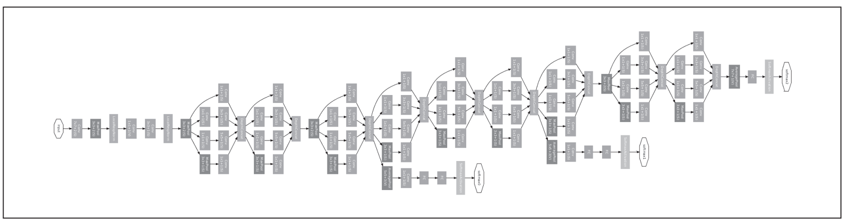


图8-10　GoogLeNet（引用自文献[23]）

只看图的话，这似乎是一个看上去非常复杂的网络结构，但实际上它基本上和之前介绍的CNN结构相同。不过，GoogLeNet的特征是，网络不仅在纵向上有深度，在横向上也有深度（广度）。

GoogLeNet在横向上有“宽度”，这称为“Inception结构”，以图8-11所示的结构为基础。

如图8-11所示，Inception结构使用了多个大小不同的滤波器（和池化），最后再合并它们的结果。GoogLeNet的特征就是将这个Inception结构用作一个构件（构成元素）。此外，在GoogLeNet中，很多地方都使用了大小为

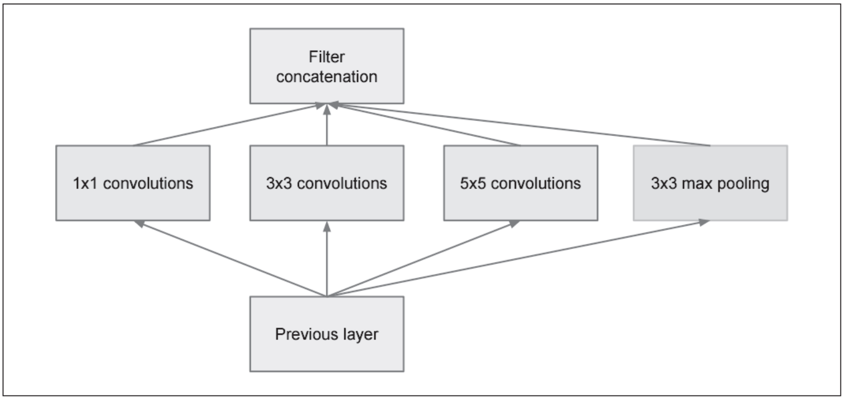


图8-11 GoogLeNet的Inception结构(引用自文献[23])

1×1的滤波器的卷积层。这个1×1的卷积运算通过在通道方向上减小大小，有助于减少参数和实现高速化处理(具体请参考原始论文[23])。

8.2.4 ResNet

ResNet[24]是微软团队开发的网络。它的特征在于具有比以前的网络更深的结构。

我们已经知道加深层对于提升性能很重要。但是，在深度学习中，过度加深层的话，很多情况下学习将不能顺利进行，导致最终性能不佳。ResNet中，为了解决这类问题，导入了“快捷结构”(也称为“捷径”或“小路”)。导入这个快捷结构后，就可以随着层的加深而不断提高性能了(当然，层的加深也是有限度的)。

如图8-12所示，快捷结构横跨(跳过)了输入数据的卷积层，将输入x合计到输出。

图8-12中，在连续2层的卷积层中，将输入x跳着连接至2层后的输出。这里的重点是，通过快捷结构，原来的2层卷积层的输出$\mathcal{F}(x)$变成了$\mathcal{F}(x)+x$。通过引入这种快捷结构，即使加深层，也能高效地学习。这是因为，通过快捷结构，反向传播时信号可以无衰减地传递。

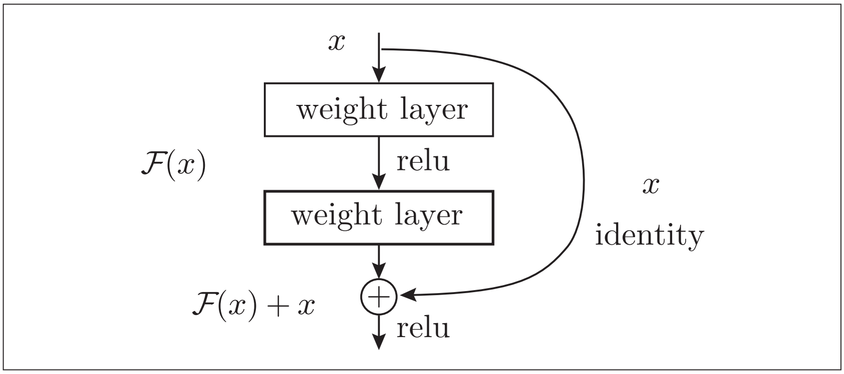


图8-12　ResNet的构成要素（引用自文献[24]）：这里的“weight layer”是指卷积层



因为快捷结构只是原封不动地传递输入数据，所以反向传播时会将来自上游的梯度原封不动地传向下游。这里的重点是不对来自上游的梯度进行任何处理，将其原封不动地传向下游。因此，基于快捷结构，不用担心梯度会变小（或变大），能够向前一层传递“有意义的梯度”。通过这个快捷结构，之前因为加深层而导致的梯度变小的梯度消失问题就有望得到缓解。

ResNet以前面介绍过的VGG网络为基础，引入快捷结构以加深层，其结果如图8-13所示。

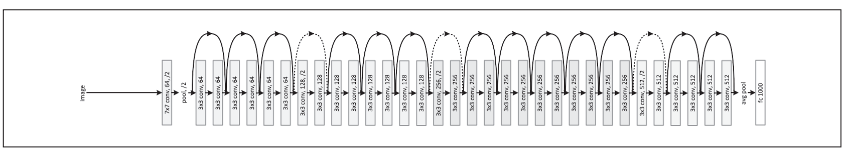


图8-13　ResNet（引用自文献[24]）：方块对应3×3的卷积层，其特征在于引入了横跨层的快捷结构

如图8-13所示，ResNet通过以2个卷积层为间隔跳跃式地连接来加深层。另外，根据实验的结果，即便加深到150层以上，识别精度也会持续提高。并且，在ILSVRC大赛中，ResNet的错误识别率为3.5%（前5类中包含正确解这一精度下的错误识别率），令人称奇。

实践中经常会灵活应用使用ImageNet这个巨大的数据集学习到的权重数据，这称为**迁移学习**，将学习完的权重（的一部分）复制到其他神经网络，进行再学习（fine tuning）。比如，准备一个和VGG相同结构的网络，把学习完的权重作为初始值，以新数据集为对象，进行再学习。迁移学习在手头数据集较少时非常有效。

8.3 深度学习的高速化

随着大数据和网络的大规模化，深度学习需要进行大量的运算。虽然到目前为止，我们都是使用CPU进行计算的，但现实是只用CPU来应对深度学习无法令人放心。实际上，环视一下周围，大多数深度学习的框架都支持**GPU**（Graphics Processing Unit），可以高速地处理大量的运算。另外，最近的框架也开始支持多个GPU或多台机器上的分布式学习。本节我们将焦点放在深度学习的计算的高速化上，然后逐步展开。深度学习的实现在8.1节就结束了，本节要讨论的高速化（支持GPU等）并不进行实现。

8.3.1 需要努力解决的问题

在介绍深度学习的高速化之前，我们先来看一下深度学习中什么样的处理比较耗时。图8-14中以AlexNet的`forward`处理为对象，用饼图展示了各层所耗费的时间。

从图中可知，AlexNex中，大多数时间都被耗费在卷积层上。实际上，卷积层的处理时间加起来占GPU整体的95%，占CPU整体的89%！因此，如何高速、高效地进行卷积层中的运算是深度学习的一大课题。虽然图8-14是推理时的结果，不过学习时也一样，卷积层中会耗费大量时间。

正如7.2节介绍的那样，卷积层中进行的运算可以追溯至乘积累加运算。因此，深度学习的高速化的主要课题就变成了如何高速、高效地进行大量的乘积累加运算。

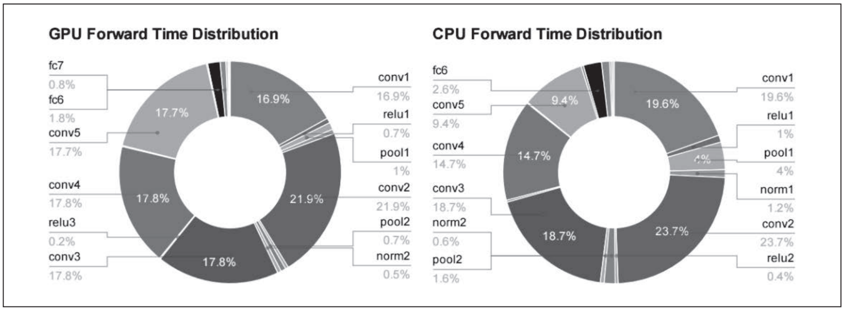


图8-14　AlexNet的forward处理中各层的时间比：左边是使用GPU的情况，右边是使用CPU的情况。图中的“conv”对应卷积层，“pool”对应池化层，“fc”对应全连接层，“norm”对应正规化层（引用自文献[26]）

8.3.2　基于GPU的高速化

GPU原本是作为图像专用的显卡使用的，但最近不仅用于图像处理，也用于通用的数值计算。由于GPU可以高速地进行并行数值计算，因此**GPU计算**的目标就是将这种压倒性的计算能力用于各种用途。所谓GPU计算，是指基于GPU进行通用的数值计算的操作。

深度学习中需要进行大量的乘积累加运算（或者大型矩阵的乘积运算）。这种大量的并行运算正是GPU所擅长的（反过来说，CPU比较擅长连续的、复杂的计算）。因此，与使用单个CPU相比，使用GPU进行深度学习的运算可以达到惊人的高速化。下面我们就来看一下基于GPU可以实现多大程度的高速化。图8-15是基于CPU和GPU进行AlexNet的学习时分别所需的时间。

从图中可知，使用CPU要花40天以上的时间，而使用GPU则可以将时间缩短至6天。此外，还可以看出，通过使用cuDNN这个最优化的库，可以进一步实现高速化。

GPU主要由NVIDIA和AMD两家公司提供。虽然两家的GPU都可以用于通用的数值计算，但与深度学习比较“亲近”的是NVIDIA的GPU。实际上，大多数深度学习框架只受益于NVIDIA的GPU。这是因为深度学习的框架中使用了NVIDIA提供的CUDA这个面向GPU计算的综合开发环境。

图8-15中出现的cuDNN是在CUDA上运行的库，它里面实现了为深度学习最优化过的函数等。

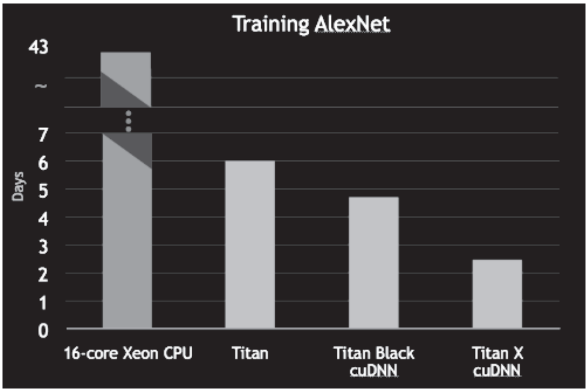


图8-15 使用CPU的“16-core Xeon CPU”和GPU的“Titan系列”进行AlexNet的学习时分别所需的时间（引用自文献[27]）



通过`im2col`可以将卷积层进行的运算转换为大型矩阵的乘积。这个`im2col`方式的实现对GPU来说是非常方便的实现方式。这是因为，相比按小规模的单位进行计算，GPU更擅长计算大规模的汇总好的数据。也就是说，通过基于`im2col`以大型矩阵的乘积的方式汇总计算，更容易发挥出GPU的能力。

8.3.3 分布式学习

虽然通过GPU可以实现深度学习运算的高速化，但即便如此，当网络较深时，学习还是需要几天到几周的时间。并且，前面也说过，深度学习伴随着很多试错。为了创建良好的网络，需要反复进行各种尝试，这样一来就必然会产生尽可能地缩短一次学习所需的时间的要求。于是，将深度学习的学习过程扩展开来的想法（也就是分布式学习）就变得重要起来。

为了进一步提高深度学习所需的计算的速度，可以考虑在多个GPU或

者多台机器上进行分布式计算。现在的深度学习框架中，出现了好几个支持多GPU或者多机器的分布式学习的框架。其中，Google的TensorFlow、微软的CNTK（Computational Network Toolki）在开发过程中高度重视分布式学习。以大型数据中心的低延迟·高吞吐网络作为支撑，基于这些框架的分布式学习呈现出惊人的效果。

基于分布式学习，可以达到何种程度的高速化呢？图8-16中显示了基于TensorFlow的分布式学习的效果。

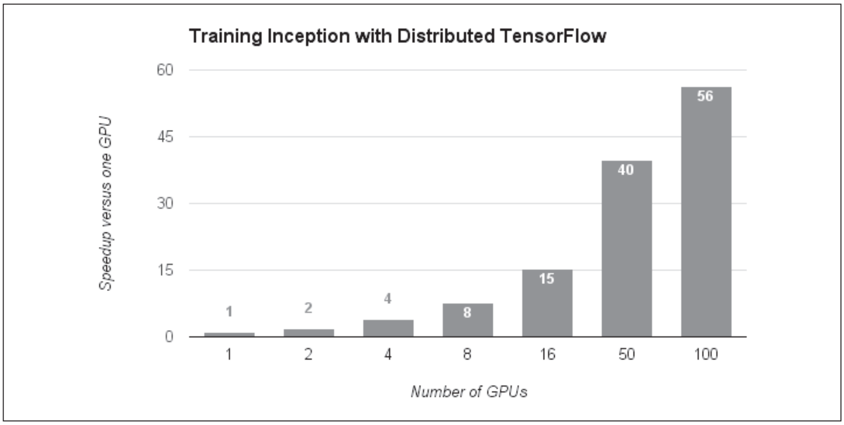


图8-16　基于TensorFlow的分布式学习的效果：横轴是GPU的个数，纵轴是与单个GPU相比时的高速化率（引用自文献[28]）

如图8-16所示，随着GPU个数的增加，学习速度也在提高。实际上，与使用1个GPU时相比，使用100个GPU（设置在多台机器上，共100个）似乎可以实现56倍的高速化！这意味着之前花费7天的学习只要3个小时就能完成，充分说明了分布式学习惊人的效果。

关于分布式学习，“如何进行分布式计算”是一个非常难的课题。它包含了机器间的通信、数据的同步等多个无法轻易解决的问题。可以将这些难题都交给TensorFlow等优秀的框架。这里，我们不讨论分布式学习的细节。关于分布式学习的技术性内容，请参考TensorFlow的技术论文（白皮书）等。

8.3.4 运算精度的位数缩减

在深度学习的高速化中，除了计算量之外，内存容量、总线带宽等也有可能成为瓶颈。关于内存容量，需要考虑将大量的权重参数或中间数据放在内存中。关于总线带宽，当流经GPU（或者CPU）总线的数据超过某个限制时，就会成为瓶颈。考虑到这些情况，我们希望尽可能减少流经网络的数据的位数。

计算机中为了表示实数，主要使用64位或者32位的浮点数。通过使用较多的位来表示数字，虽然数值计算时的误差造成的影响变小了，但计算的处理成本、内存使用量却相应地增加了，还给总线带宽带来了负荷。

关于数值精度（用几位数据表示数值），我们已经知道深度学习并不那么需要数值精度的位数。这是神经网络的一个重要性质。这个性质是基于神经网络的健壮性而产生的。这里所说的健壮性是指，比如，即便输入图像附有一些小的噪声，输出结果也仍然保持不变。可以认为，正是因为有了这个健壮性，流经网络的数据即便有所“劣化”，对输出结果的影响也较小。

计算机中表示小数时，有32位的单精度浮点数和64位的双精度浮点数等格式。根据以往的实验结果，在深度学习中，即便是16位的**半精度浮点数**（half float），也可以顺利地进行学习[30]。实际上，NVIDIA的下一代GPU框架Pascal也支持半精度浮点数的运算，由此可以认为今后半精度浮点数将被作为标准使用。

> NVIDIA的Maxwell GPU虽然支持半精度浮点数的存储（保存数据的功能），但是运算本身不是用16位进行的。下一代的Pascal框架，因为运算也是用16位进行的，所以只用半精度浮点数进行计算，就有望实现超过上一代GPU约2倍的高速化。

以往的深度学习的实现中并没有注意数值的精度，不过Python中一般使用64位的浮点数。NumPy中提供了16位的半精度浮点数类型（不过，只有16位类型的存储，运算本身不用16位进行），即便使用NumPy的半精度浮点数，识别精度也不会下降。相关的论证也很简单，有兴趣的读者请参考`ch08/half_float_network.py`。

关于深度学习的位数缩减，到目前为止已有若干研究。最近有人提出了用1位来表示权重和中间数据的Binarized Neural Networks方法[31]。为了实现深度学习的高速化，位数缩减是今后必须关注的一个课题，特别是在面向嵌入式应用程序中使用深度学习时，位数缩减非常重要。

8.4　深度学习的应用案例

前面，作为使用深度学习的例子，我们主要讨论了手写数字识别的图像类别分类问题（称为“物体识别”）。不过，深度学习并不局限于物体识别，还可以应用于各种各样的问题。此外，在图像、语音、自然语言等各个不同的领域，深度学习都展现了优异的性能。本节将以计算机视觉这个领域为中心，介绍几个深度学习能做的事情（应用）。

8.4.1　物体检测

物体检测是从图像中确定物体的位置，并进行分类的问题。如图8-17所示，要从图像中确定物体的种类和物体的位置。



图8-17　物体检测的例子（引用自文献[34]）

观察图8-17可知，物体检测是比物体识别更难的问题。之前介绍的物体识别是以整个图像为对象的，但是物体检测需要从图像中确定类别的位置，而且还有可能存在多个物体。

对于这样的物体检测问题，人们提出了多个基于CNN的方法。这些方法展示了非常优异的性能，并且证明了在物体检测的问题上，深度学习是非常有效的。

在使用CNN进行物体检测的方法中，有一个叫作R-CNN[35]的有名的方法。图8-18显示了R-CNN的处理流。

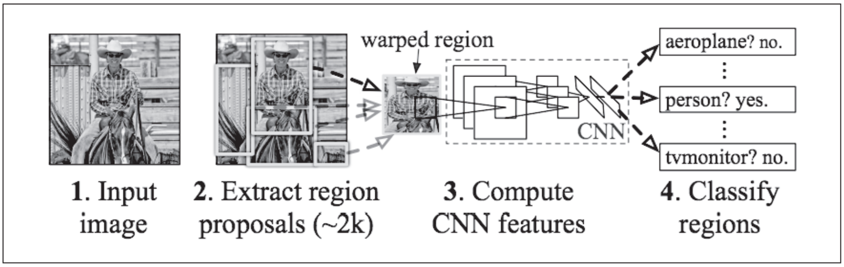


图8-18　R-CNN的处理流（引用自文献[35]）

希望大家注意图中的“2.Extract region proposals”（候选区域的提取）和“3.Compute CNN features”（CNN特征的计算）的处理部分。这里，首先（以某种方法）找出形似物体的区域，然后对提取出的区域应用CNN进行分类。R-CNN中会将图像变形为正方形，或者在分类时使用SVM（支持向量机），实际的处理流会稍微复杂一些，不过从宏观上看，也是由刚才的两个处理（候选区域的提取和CNN特征的计算）构成的。

在R-CNN的前半部分的处理——候选区域的提取（发现形似目标物体的处理）中，可以使用计算机视觉领域积累的各种各样的方法。R-CNN的论文中使用了一种被称为Selective Search的方法，最近还提出了一种基于CNN来进行候选区域提取的Faster R-CNN方法[36]。Faster R-CNN用一个CNN来完成所有处理，使得高速处理成为可能。

8.4.2　图像分割

图像分割是指在像素水平上对图像进行分类。如图8-19所示，使用以像素为单位对各个对象分别着色的监督数据进行学习。然后，在推理时，对输入图像的所有像素进行分类。



图8-19　图像分割的例子（引用自文献[34]）：左边是输入图像，右边是监督用的带标签图像

之前实现的神经网络是对图像整体进行了分类，要将它落实到像素水平的话，该怎么做呢？

要基于神经网络进行图像分割，最简单的方法是以所有像素为对象，对每个像素执行推理处理。比如，准备一个对某个矩形区域中心的像素进行分类的网络，以所有像素为对象执行推理处理。正如大家能想到的，这样的方法需要按照像素数量进行相应次`forward`处理，因而需要耗费大量的时间（正确地说，卷积运算中会发生重复计算很多区域的无意义的计算）。为了解决这个无意义的计算问题，有人提出了一个名为FCN（Fully Convolutional Network）[37]的方法。该方法通过一次`forward`处理，对所有像素进行分类（图8-20）。

FCN的字面意思是“全部由卷积层构成的网络”。相对于一般的CNN包含全连接层，FCN将全连接层替换成发挥相同作用的卷积层。在物体识别中使用的网络的全连接层中，中间数据的空间容量被作为排成一列的节点进

行处理，而只由卷积层构成的网络中，空间容量可以保持原样直到最后的输出。

如图8-20所示，FCN的特征在于最后导入了扩大空间大小的处理。基于这个处理，变小了的中间数据可以一下子扩大到和输入图像一样的大小。FCN最后进行的扩大处理是基于双线性插值法的扩大（双线性插值扩大）。FCN中，这个双线性插值扩大是通过去卷积（逆卷积运算）来实现的（细节请参考FCN的论文[37]）。

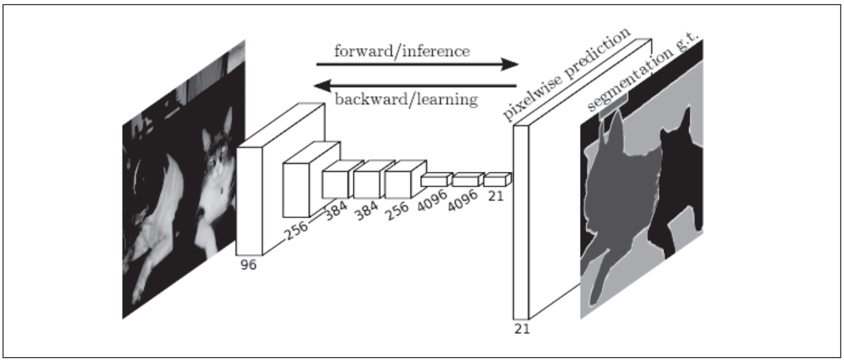


图8-20　FCN的概略图（引用自文献[37]）

全连接层中，输出和全部的输入相连。使用卷积层也可以实现与此结构完全相同的连接。比如，针对输入大小是32×10×10（通道数32、高10、长10）的数据的全连接层可以替换成滤波器大小为32×10×10的卷积层。如果全连接层的输出节点数是100，那么在卷积层准备100个32×10×10的滤波器就可以实现完全相同的处理。像这样，全连接层可以替换成进行相同处理的卷积层。

8.4.3　图像标题的生成

有一项融合了计算机视觉和自然语言的有趣的研究，该研究如图8-21所示，给出一个图像后，会自动生成介绍这个图像的文字（图像的标题）。

给出一个图像后，会像图8-21一样自动生成表示该图像内容的文本。比如，左上角的第一幅图像生成了文本“A person riding a motorcycle on a



图8-21　基于深度学习的图像标题生成的例子（引用自文献[38]）

dirt road.”(在没有铺装的道路上骑摩托车的人)，而且这个文本只从该图像自动生成。文本的内容和图像确实是一致的。并且，令人惊讶的是，除了“骑摩托车”之外，连“没有铺装的道路”都被正确理解了。

一个基于深度学习生成图像标题的代表性方法是被称为NIC(Neural Image Caption)的模型。如图8-22所示，NIC由深层的CNN和处理自然语言的RNN(Recurrent Neural Network)构成。RNN是呈递归式连接的网络，经常被用于自然语言、时间序列数据等连续性的数据上。

NIC基于CNN从图像中提取特征，并将这个特征传给RNN。RNN以CNN提取出的特征为初始值，递归地生成文本。这里，我们不深入讨论技术上的细节，不过基本上NIC是组合了两个神经网络(CNN和RNN)的简单结构。基于NIC，可以生成惊人的高精度的图像标题。我们将组合图像和自然语言等多种信息进行的处理称为**多模态处理**。多模态处理是近年来备受关注的一个领域。

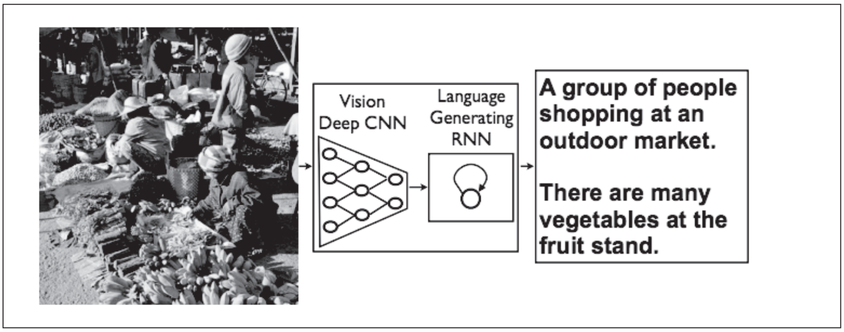


图8-22 **Neural Image Caption（NIC）的整体结构（引用自文献[38]）**



RNN的R表示Recurrent（递归的）。这个递归指的是神经网络的递归的网络结构。根据这个递归结构，神经网络会受到之前生成的信息的影响（换句话说，会记忆过去的信息），这是RNN的特征。比如，生成“我”这个词之后，下一个要生成的词受到“我”这个词的影响，生成了“要”；然后，再受到前面生成的“我要”的影响，生成了“睡觉”这个词。对于自然语言、时间序列数据等连续性的数据，RNN以记忆过去的信息的方式运行。

8.5 深度学习的未来

深度学习已经不再局限于以往的领域，开始逐渐应用于各个领域。本节将介绍几个揭示了深度学习的可能性和未来的研究。

8.5.1 图像风格变换

有一项研究是使用深度学习来“绘制”带有艺术气息的画。如图8-23所示，输入两个图像后，会生成一个新的图像。两个输入图像中，一个称为“内容图像”，另一个称为“风格图像”。

如图8-23所示，如果指定将梵高的绘画风格应用于内容图像，深度学习就会按照指示绘制出新的画作。此项研究出自论文“A Neural Algorithm of

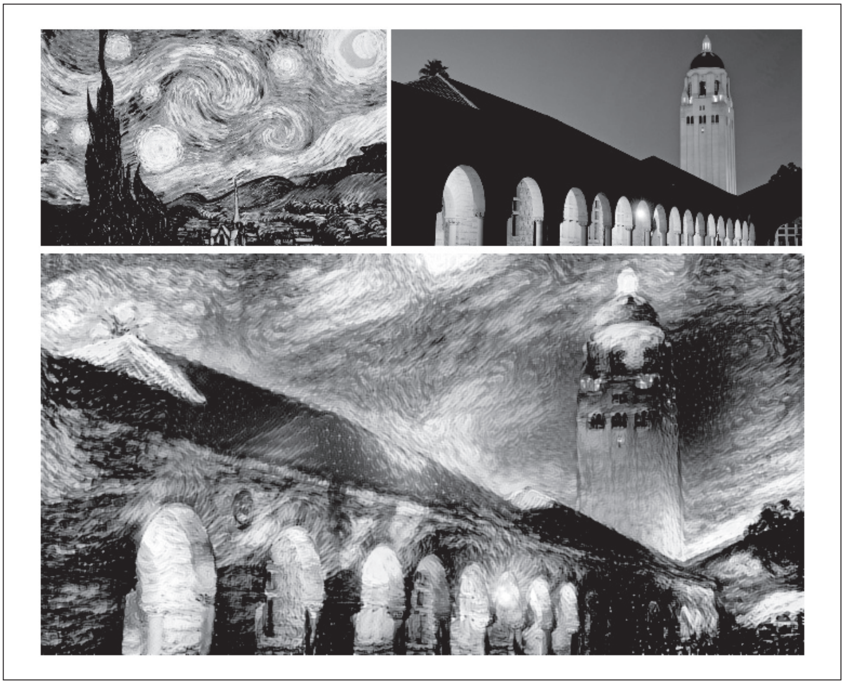


图 8-23 基于论文“A Neural Algorithm of Artistic Style”的图像风格变换的例子：左上角是风格图像，右上角是内容图像，下面的图像是新生成的图像（图像引用自文献 [40]）

Artistic Style”[39]，一经发表就受到全世界的广泛关注。

这里我们不会介绍这项研究的详细内容，只是叙述一下这个技术的大致框架，即刚才的方法是在学习过程中使网络的中间数据近似内容图像的中间数据。这样一来，就可以使输入图像近似内容图像的形状。此外，为了从风格图像中吸收风格，导入了风格矩阵的概念。通过在学习过程中减小风格矩阵的偏差，就可以使输入图像接近梵高的风格。

8.5.2 图像的生成

刚才的图像风格变换的例子在生成新的图像时输入了两个图像。不同于这种研究，现在有一种研究是生成新的图像时不需要任何图像（虽然需要事

先使用大量的图像进行学习，但在“画”新图像时不需要任何图像)。比如，基于深度学习，可以实现从零生成“卧室”的图像。图8-24中展示的图像是基于**DCGAN**(Deep Convolutional Generative Adversarial Network)[41]方法生成的卧室图像的例子。



图8-24 基于DCGAN生成的新的卧室图像(引用自文献[41])

图8-24的图像可能看上去像是真的照片，但其实这些图像都是基于DCGAN新生成的图像。也就是说，DCGAN生成的图像是谁都没有见过的图像(学习数据中没有的图像)，是从零生成的新图像。

能画出以假乱真的图像的DCGAN会将图像的生成过程模型化。使用大量图像(比如，印有卧室的大量图像)训练这个模型，学习结束后，使用这个模型，就可以生成新的图像。

DCGAN中使用了深度学习，其技术要点是使用了Generator(生成者)和Discriminator(识别者)这两个神经网络。Generator生成近似真品的图像，Discriminator判别它是不是真图像(是Generator生成的图像还是实际拍摄的图像)。像这样，通过让两者以竞争的方式学习，Generator会学习到更加精妙的图像作假技术，Discriminator则会成长为能以更高精度辨别真假的鉴定师。两者互相切磋、共同成长，这是**GAN**(Generative Adversarial

Network）这个技术的有趣之处。在这样的切磋中成长起来的Generator最终会掌握画出足以以假乱真的图像的能力（或者说有这样的可能）。

> 之前我们见到的机器学习问题都是被称为**监督学习**（supervised learning）的问题。这类问题就像手写数字识别一样，使用的是图像数据和教师标签成对给出的数据集。不过这里讨论的问题，并没有给出监督数据，只给了大量的图像（图像的集合），这样的问题称为**无监督学习**（unsupervised learning）。无监督学习虽然是很早之前就开始研究的领域（Deep Belief Network、Deep Boltzmann Machine等很有名），但最近似乎并不是很活跃。今后，随着使用深度学习的DCGAN等方法受到关注，无监督学习有望得到进一步发展。

8.5.3 自动驾驶

计算机代替人类驾驶汽车的自动驾驶技术有望得到实现。除了汽车制造商之外，IT企业、大学、研究机构等也都在为实现自动驾驶而进行着激烈的竞争。自动驾驶需要结合各种技术的力量来实现，比如决定行驶路线的路线计划（path plan）技术、照相机或激光等传感技术等，在这些技术中，正确识别周围环境的技术据说尤其重要。这是因为要正确识别时刻变化的环境、自由来往的车辆和行人是非常困难的。

如果可以在各种环境中稳健地正确识别行驶区域的话，实现自动驾驶可能也就没那么遥远了。最近，在识别周围环境的技术中，深度学习的力量备受期待。比如，基于CNN的神经网络SegNet[42]，可以像图8-25那样高精度地识别行驶环境。

图8-25中对输入图像进行了分割（像素水平的判别）。观察结果可知，在某种程度上正确地识别了道路、建筑物、人行道、树木、车辆等。今后若能基于深度学习使这种技术进一步实现高精度化、高速化的话，自动驾驶的实用化可能也就没那么遥远了。



图8-25 基于深度学习的图像分割的例子：道路、车辆、建筑物、人行道等被高精度地识别了出来（引用自文献[43]）

8.5.4 Deep Q-Network（强化学习）

就像人类通过摸索试验来学习一样（比如骑自行车），让计算机也在摸索试验的过程中自主学习，这称为**强化学习**（reinforcement learning）。强化学习和有"教师"在身边教的"监督学习"有所不同。

强化学习的基本框架是，代理（Agent）根据环境选择行动，然后通过这个行动改变环境。根据环境的变化，代理获得某种报酬。强化学习的目的是决定代理的行动方针，以获得更好的报酬（图8-26）。

图8-26中展示了强化学习的基本框架。这里需要注意的是，报酬并不是确定的，只是"预期报酬"。比如，在《超级马里奥兄弟》这款电子游戏中，让马里奥向右移动能获得多少报酬不一定是明确的。这时需要从游戏得分（获得的硬币、消灭的敌人等）或者游戏结束等明确的指标来反向计算，决定"预期报酬"。如果是监督学习的话，每个行动都可以从"教师"那里获得正确的评价。

在使用了深度学习的强化学习方法中，有一个叫作Deep Q-Network（通称**DQN**）[44]的方法。该方法基于被称为Q学习的强化学习算法。这里省略

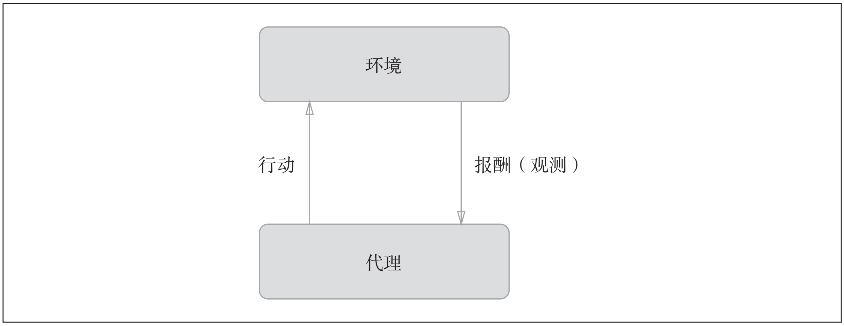


图8-26 强化学习的基本框架：代理自主地进行学习，以获得更好的报酬

Q学习的细节，不过在Q学习中，为了确定最合适的行动，需要确定一个被称为最优行动价值函数的函数。为了近似这个函数，DQN使用了深度学习（CNN）。

在DQN的研究中，有让电子游戏自动学习，并实现了超过人类水平的操作的例子。如图8-27所示，DQN中使用的CNN把游戏图像的帧（连续4帧）作为输入，最终输出游戏手柄的各个动作（控制杆的移动量、按钮操作的有无等）的“价值”。

之前在学习电子游戏时，一般是把游戏的状态（人物的地点等）事先提取出来，作为数据给模型。但是，在DQN中，如图8-27所示，输入数据只有电子游戏的图像。这是DQN值得大书特书的地方，可以说大幅提高了DQN的实用性。为什么呢？因为这样就无需根据每个游戏改变设置，只要给DQN游戏图像就可以了。实际上，DQN可以用相同的结构学习《吃豆人》、*Atari*等很多游戏，甚至在很多游戏中取得了超过人类的成绩。



人工智能AlphaGo[45]击败围棋冠军的新闻受到了广泛关注。这个AlphaGo技术的内部也用了深度学习和强化学习。AlphaGo学习了3000万个专业棋手的棋谱，并且不停地重复自己和自己的对战，积累了大量的学习经验。AlphaGo和DQN都是Google的Deep Mind公司进行的研究，该公司今后的研究值得密切关注。

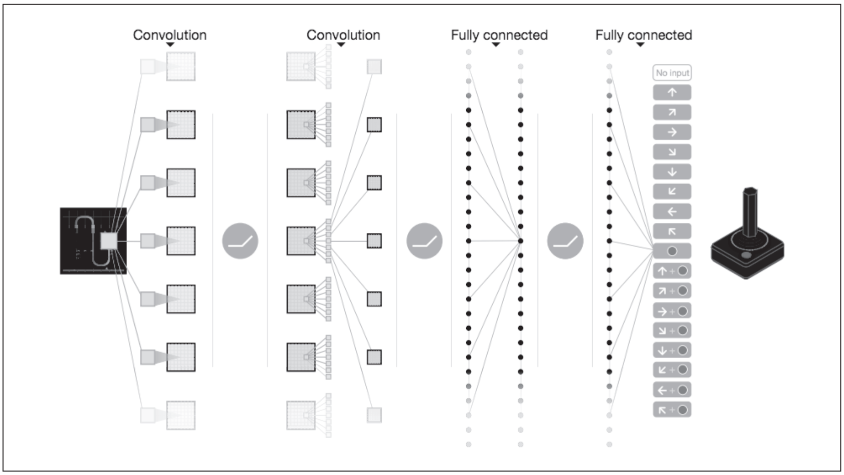


图8-27 基于Deep Q-Network学习电子游戏的操作。输入是电子游戏的图像，经过摸索试验，学习出让专业玩家都自愧不如的游戏手柄（操作杆）的操作手法（引用自文献[44]）

8.6 小结

本章我们实现了一个（稍微）深层的CNN，并在手写数字识别上获得了超过99%的高识别精度。此外，还讲解了加深网络的动机，指出了深度学习在朝更深的方向前进。之后，又介绍了深度学习的趋势和应用案例，以及对高速化的研究和代表深度学习未来的研究案例。

深度学习领域还有很多尚未揭晓的东西，新的研究正一个接一个地出现。今后，全世界的研究者和技术专家也将继续积极从事这方面的研究，一定能实现目前无法想象的技术。

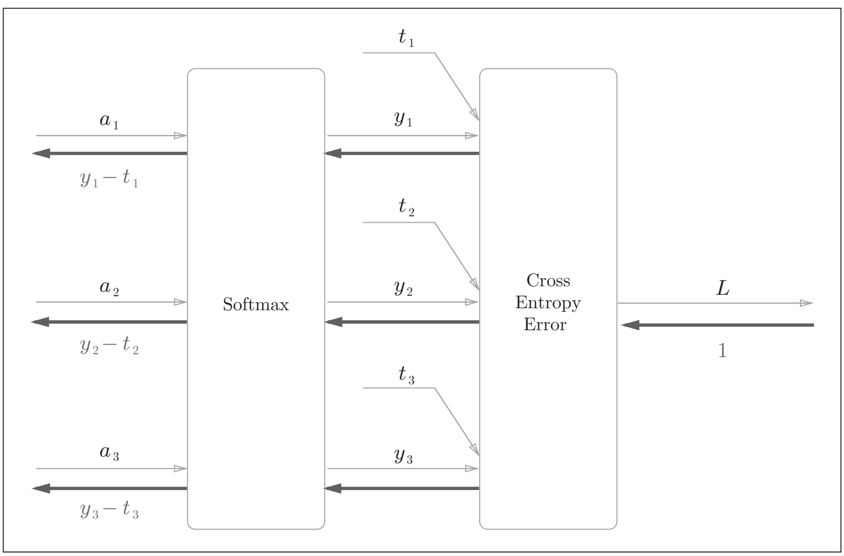
感谢读者一直读到本书的最后一章。如果读者能通过本书加深对深度学习的理解，体会到深度学习的有趣之处，笔者将深感荣幸。

本章所学的内容

- 对于大多数的问题，都可以期待通过加深网络来提高性能。
- 在最近的图像识别大赛 ILSVRC 中，基于深度学习的方法独占鳌头，使用的网络也在深化。
- VGG、GoogLeNet、ResNet 等是几个著名的网络。
- 基于 GPU、分布式学习、位数精度的缩减，可以实现深度学习的高速化。
- 深度学习（神经网络）不仅可以用于物体识别，还可以用于物体检测、图像分割。
- 深度学习的应用包括图像标题的生成、图像的生成、强化学习等。最近，深度学习在自动驾驶上的应用也备受期待。

附录A Softmax-with-Loss层的计算图

这里，我们给出softmax函数和交叉熵误差的计算图，来求它们的反向传播。softmax函数称为softmax层，交叉熵误差称为Cross Entropy Error层，两者的组合称为Softmax-with-Loss层。先来看一下结果，Softmax-with-Loss层可以画成图A-1所示的计算图。



图A-1 Softmax-with-Loss层的计算图

图A-1的计算图中假定了一个进行3类别分类的神经网络。从前面的层输入的是(a_1, a_2, a_3)，softmax层输出(y_1, y_2, y_3)。此外，教师标签是(t_1, t_2, t_3)，Cross Entropy Error层输出损失L。

如图A-1所示，在本附录中，Softmac-with-Loss层的反向传播的结果为$(y_1 - t_1, y_2 - t_2, y_3 - t_3)$。

A.1 正向传播

图A-1的计算图中省略了Softmax层和Cross Entropy Error层的内容。这里，我们来画出这两个层的内容。

首先是Softmax层。softmax函数可由下式表示。

$$y_k = \frac{\exp(a_k)}{\sum\limits_{i=1}^{n} \exp(a_i)} \tag{A.1}$$

因此，用计算图表示Softmax层的话，则如图A-2所示。

图A-2的计算图中，指数的和（相当于式(A.1)的分母）简写为S，最终的输出记为(y_1, y_2, y_3)。

接下来是Cross Entropy Error层。交叉熵误差可由下式表示。

$$L = -\sum_k t_k \log y_k \tag{A.2}$$

根据式(A.2)，Cross Entropy Error层的计算图可以画成图A-3那样。

图A-3的计算图很直观地表示出了式(A.2)，所以应该没有特别难的地方。

下一节，我们将看一下反向传播。

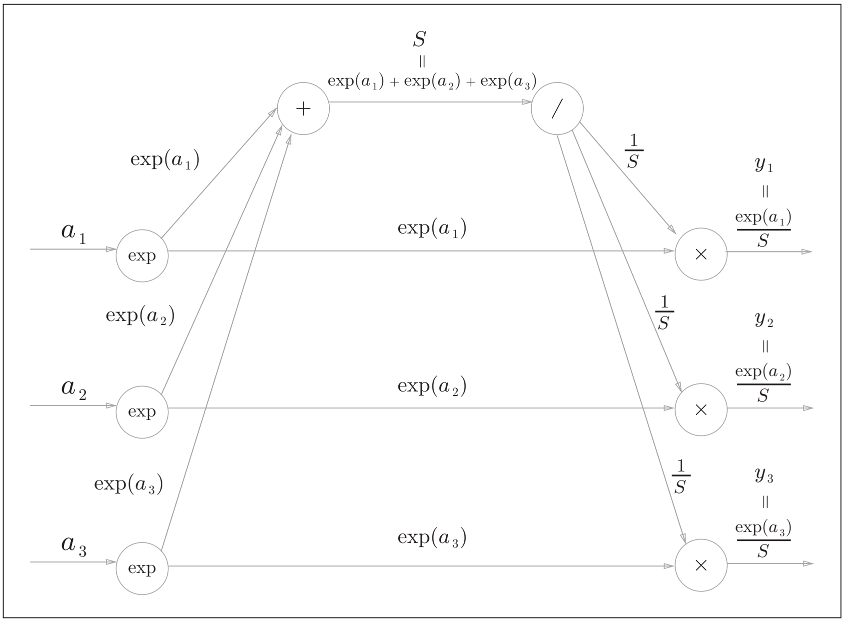


图 A-2　Softmax 层的计算图（仅正向传播）

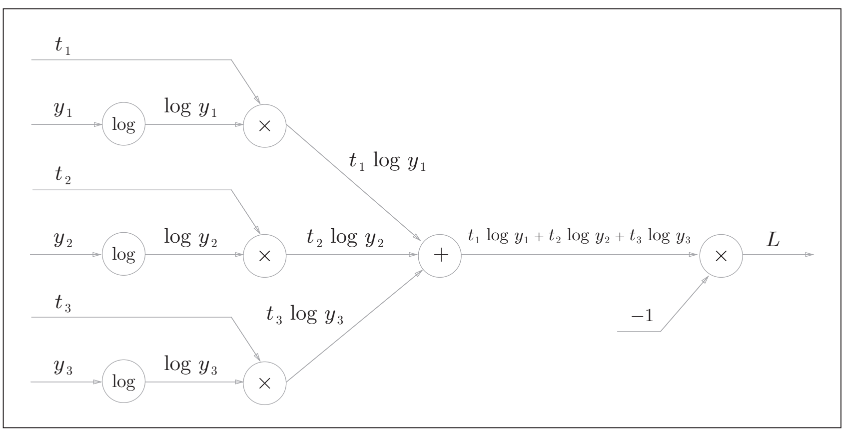
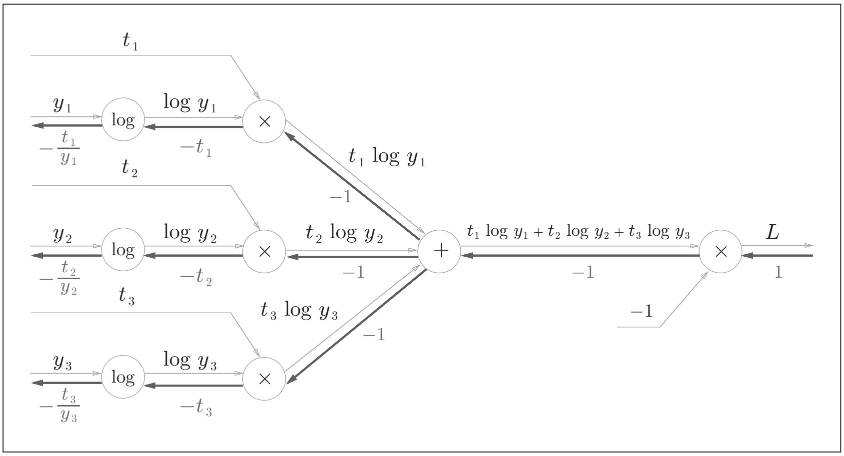


图 A-3　Cross Entropy Error 层的计算图（仅正向传播）

A.2 反向传播

首先是Cross Entropy Error层的反向传播。Cross Entropy Error层的反向传播可以画成图A-4那样。



图A-4 交叉熵误差的反向传播

求这个计算图的反向传播时，要注意下面几点。

- 反向传播的初始值(图A-4中最右边的值)是1(因为 $\frac{\partial L}{\partial L} = 1$)。
- “×”节点的反向传播将正向传播时的输入值翻转，乘以上游传过来的导数后，再传给下游。
- “+”节点将上游传来的导数原封不动地传给下游。
- “log”节点的反向传播遵从下式。

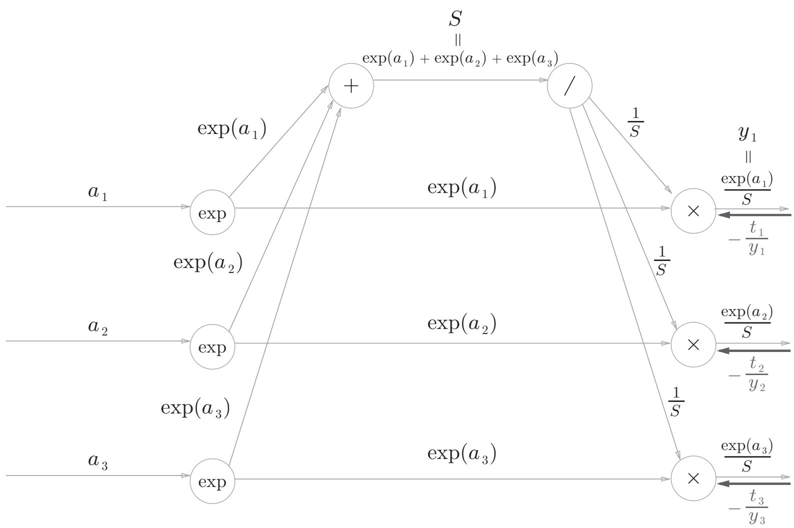
$$y = \log x$$

$$\frac{\partial y}{\partial x} = \frac{1}{x}$$

遵从以上几点，就可以轻松求得Cross Entropy Error的反向传播。结果$\left(-\frac{t_1}{y_1}, -\frac{t_2}{y_2}, -\frac{t_3}{y_3}\right)$是传给Softmax层的反向传播的输入。

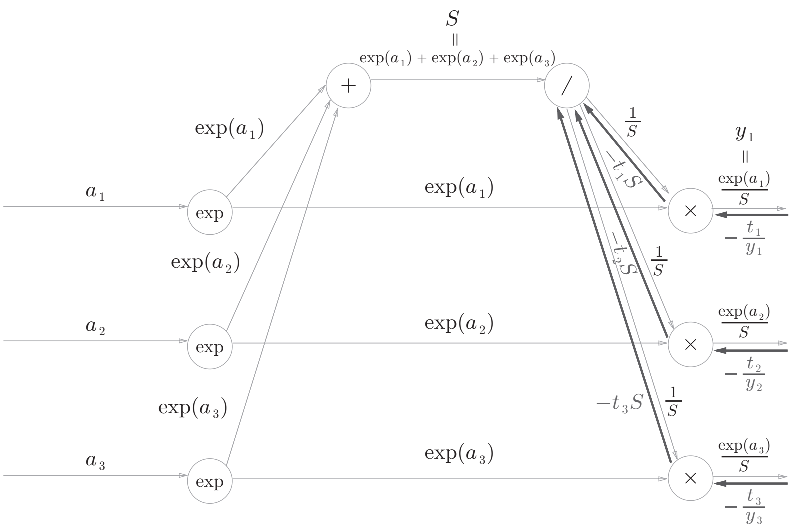
下面是Softmax层的反向传播的步骤。因为Softmax层有些复杂，所以我们来逐一进行确认。

步骤1



前面的层（Cross Entropy Error层）的反向传播的值传过来。

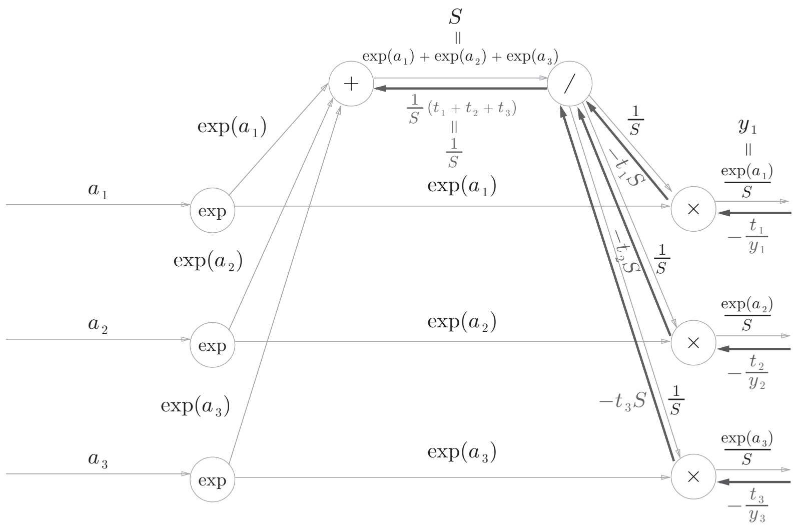
步骤2



“×”节点将正向传播的值翻转后相乘。这个过程中会进行下面的计算。

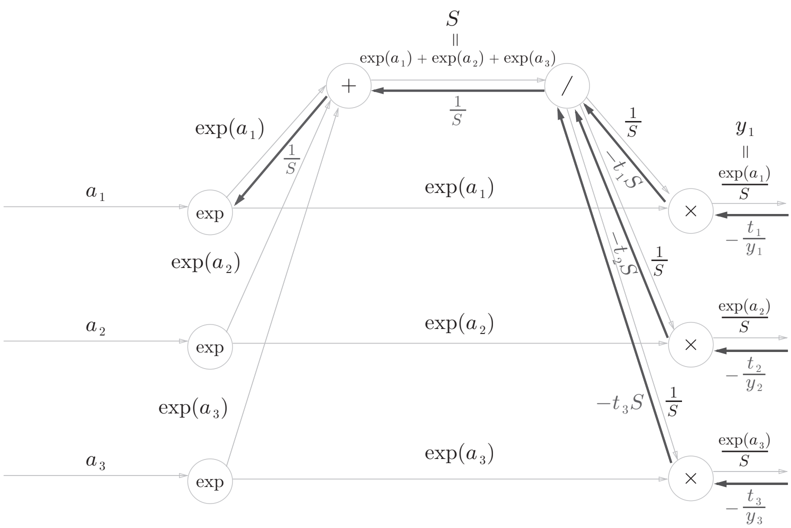
$$-\frac{t_1}{y_1}\exp(a_1) = -t_1\frac{S}{\exp(a_1)}\exp(a_1) = -t_1 S \qquad \text{(A.3)}$$

步骤3



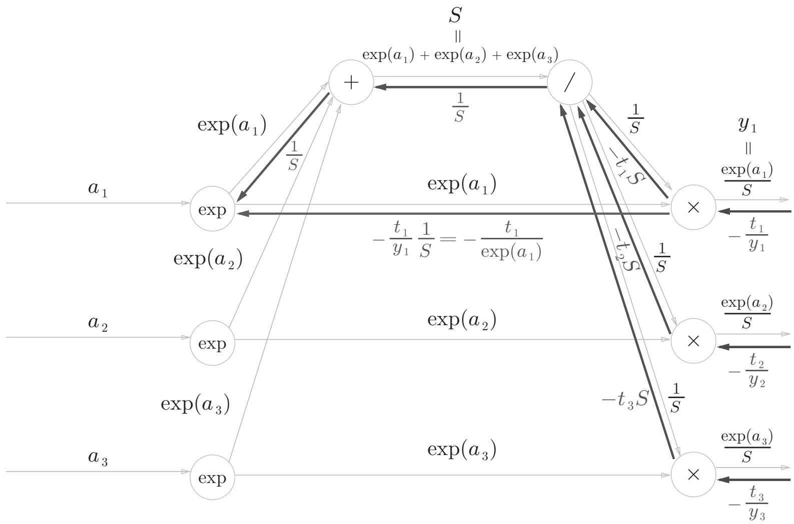
正向传播时若有分支流出，则反向传播时它们的反向传播的值会相加。因此，这里分成了三支的反向传播的值 $(-t_1S, -t_2S, -t_3S)$ 会被求和。然后，还要对这个相加后的值进行"/"节点的反向传播，结果为 $\frac{1}{S}(t_1+t_2+t_3)$。这里，(t_1, t_2, t_3) 是教师标签，也是one-hot向量。one-hot向量意味着 (t_1, t_2, t_3) 中只有一个元素是1，其余都是0。因此，(t_1, t_2, t_3) 的和为1。

步骤4



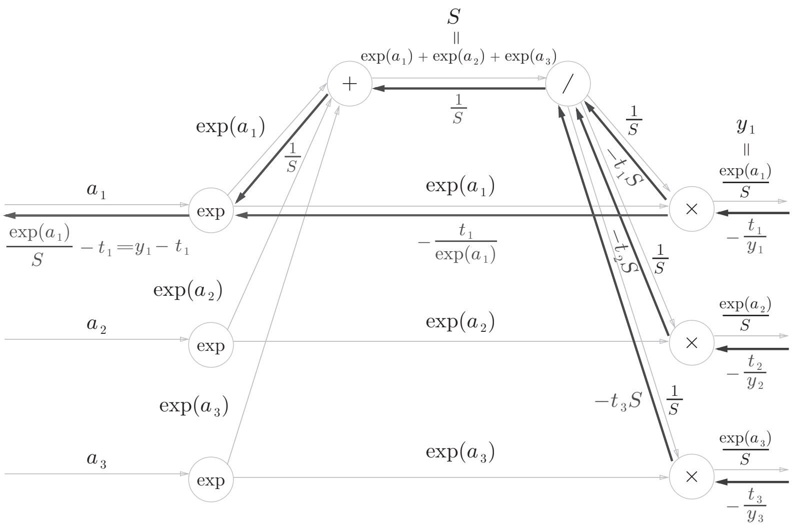
"+"节点原封不动地传递上游的值。

步骤 5



"×"节点将值翻转后相乘。这里，式子变形时使用了 $y_1=\frac{\exp(a_1)}{S}$。

步骤6



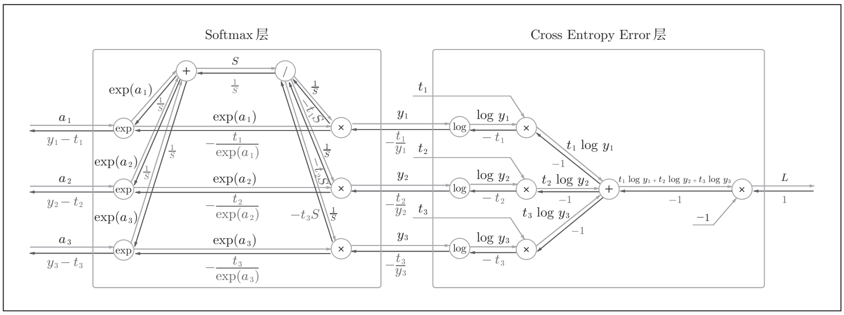
"exp"节点中有下面的关系式成立。

$$
\begin{aligned}
y &= \exp(x) \\
\frac{\partial y}{\partial x} &= \exp(x)
\end{aligned}
\tag{A.4}
$$

根据这个式子，向两个分支的输入和乘以$\exp(a_1)$后的值就是我们要求的反向传播。用式子写出来的话，就是$\left(\frac{1}{S} - \frac{t_1}{\exp(a_1)}\right)\exp(a_1)$，整理之后为$y_1 - t_1$。综上，我们推导出，正向传播时输入是$a_1$的节点，它的反向传播是$y_1 - t_1$。剩下的$a_2$、$a_3$也可以按照相同的步骤求出来(结果分别为$y_2 - t_2$和$y_3 - t_3$)。此外，除了这里介绍的3类别分类外，对于$n$类别分类的情况，也可以推导出同样的结果。

A.3 小结

上面，我们画出了Softmax-with-Loss层的计算图的全部内容，并求了它的反向传播。未做省略的Softmax-with-Loss层的计算图如图A-5所示。



图A-5 Softmax-with-Loss层的计算图

图A-5的计算图看上去很复杂，但是使用计算图逐个确认的话，求导(反向传播的步骤)也并没有那么复杂。除了这里介绍的Softmax-with-Loss层，遇到其他看上去很难的层(如Batch Normalization层)时，请一定按照这里的步骤思考一下。相信会比只看数学式更容易理解。

参考文献

Python / NumPy

[1] Bill Lubanovic. Introducing Python①. O'Reilly Media, 2014.

[2] Wes McKinney. Python for Data Analysis②. O'Reilly Media.

[3] Scipy Lecture Notes.

计算图（误差反向传播法）

[4] Andrej Karpathy's blog "Hacker's guide to Neural Networks".

深度学习的在线课程（资料）

[5] CS231n: Convolutional Neural Networks for Visual Recognition.

参数的更新方法

[6] John Duchi, Elad Hazan, and Yoram Singer(2011): Adaptive Subgradient Methods for Online Learning and Stochastic Optimization.

① 中文版名为《Python语言及其应用》，梁杰等译，人民邮电出版社2015年出版。——编者注

② 中文版名为《利用Python进行数据分析》，唐学韬译，机械工业出版社2013年出版。——编者注

Journal of Machine Learning Research 12, Jul (2011), 2121 – 2159.

[7] Tieleman, T., & Hinton, G.(2012): Lecture 6.5—RMSProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning.

[8] Diederik Kingma and Jimmy Ba.(2014): Adam: A Method for Stochastic Optimization. arXiv:1412.6980 [cs] (December 2014).

权重参数的初始值

[9] Xavier Glorot and Yoshua Bengio(2010): Understanding the difficulty of training deep feedforward neural networks. In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS2010). Society for Artificial Intelligence and Statistics.

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun(2015): Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. In 1026 – 1034.

Batch Normalization / Dropout

[11] Sergey Ioffe and Christian Szegedy(2015): Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. arXiv:1502.03167 [cs] (February 2015).

[12] Dmytro Mishkin and Jiri Matas(2015): All you need is a good init. arXiv:1511.06422 [cs] (November 2015).

[13] Frederik Kratzert's blog "Understanding the backward pass through Batch Normalization Layer".

[14] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov(2014): Dropout: A simple way to prevent neural

networks from overfitting. The Journal of Machine Learning Research, pages 1929 – 1958, 2014.

超参数的最优化

[15] James Bergstra and Yoshua Bengio (2012): Random Search for Hyper-Parameter Optimization. Journal of Machine Learning Research 13, Feb (2012), 281 – 305.

[16] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams (2012): Practical Bayesian Optimization of Machine Learning Algorithms. In F. Pereira, C. J. C. Burges, L. Bottou, & K. Q. Weinberger, eds. Advances in Neural Information Processing Systems 25. Curran Associates, Inc., 2951 – 2959.

CNN的可视化

[17] Matthew D. Zeiler and Rob Fergus (2014): Visualizing and Understanding Convolutional Networks. In David Fleet, Tomas Pajdla, Bernt Schiele, & Tinne Tuytelaars, eds. Computer Vision – ECCV 2014. Lecture Notes in Computer Science. Springer International Publishing, 818 – 833.

[18] A. Mahendran and A. Vedaldi (2015): Understanding deep image representations by inverting them. In 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). 5188 – 5196.

[19] Donglai Wei, Bolei Zhou, Antonio Torralba, William T. Freeman (2015): mNeuron: A Matlab Plugin to Visualize Neurons from Deep Models.

具有代表性的网格

[20] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner (1998): Gradient-based learning applied to document recognition. Proceedings of the IEEE 86, 11 (November 1998), 2278 – 2324.

[21] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton (2012): ImageNet Classification with Deep Convolutional Neural Networks. In F. Pereira, C. J. C. Burges, L. Bottou, & K. Q. Weinberger, eds. Advances in Neural Information Processing Systems 25. Curran Associates, Inc., 1097 – 1105.

[22] Karen Simonyan and Andrew Zisserman (2014): Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv:1409.1556 [cs] (September 2014).

[23] Christian Szegedy et al (2015): Going Deeper With Convolutions. In The IEEE Conference on Computer Vision and Pattern Recognition (CVPR).

[24] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun (2015): Deep Residual Learning for Image Recognition. arXiv:1512.03385 [cs] (December 2015).

数据集

[25] J. Deng, W. Dong, R. Socher, L.J. Li, Kai Li, and Li Fei-Fei (2009): ImageNet: A large-scale hierarchical image database. In IEEE Conference on Computer Vision and Pattern Recognition, 2009. CVPR 2009. 248 – 255.

计算的高速化

[26] Jia Yangqing(2014): Learning Semantic Image Representations at a Large Scale. PhD thesis, EECS Department, University of California, Berkeley, May 2014.

[27] NVIDIA blog “NVIDIA Propels Deep Learning with TITAN X, New DIGITS Training System and DevBox”.

[28] Google Research Blog “Announcing TensorFlow 0.8 – now with distributed computing support!”.

[29] Martín Abadi et al(2016): TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. arXiv:1603.04467 [cs] (March 2016).

[30] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan(2015): Deep learning with limited numerical precision. CoRR, abs/1502.02551 392 (2015).

[31] Matthieu Courbariaux and Yoshua Bengio(2016): Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. arXiv preprint arXiv:1602.02830 (2016).

MNIST数据集识别精度排行榜及最高精度的方法

[32] Rodrigo Benenson’s blog “Classification datasets results”.

[33] Li Wan, Matthew Zeiler, Sixin Zhang, Yann L. Cun, and Rob Fergus (2013): Regularization of Neural Networks using DropConnect. In Sanjoy Dasgupta & David McAllester, eds. Proceedings of the 30th International Conference on Machine Learning (ICML2013). JMLR Workshop and Conference Proceedings, 1058 – 1066.

深度学习的应用

[34] Visual Object Classes Challenge 2012 VO(2012).

[35] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik(2014): Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation. In 580 – 587.

[36] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun(2015): Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, & R. Garnett, eds. Advances in Neural Information Processing Systems 28. Curran Associates, Inc., 91 – 99.

[37] Jonathan Long, Evan Shelhamer, and Trevor Darrell(2015): Fully Convolutional Networks for Semantic Segmentation. In The IEEE Conference on Computer Vision and Pattern Recognition (CVPR).

[38] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan (2015): Show and Tell: A Neural Image Caption Generator. In The IEEE Conference on Computer Vision and Pattern Recognition (CVPR).

[39] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge(2015): A Neural Algorithm of Artistic Style. arXiv:1508.06576 [cs, q-bio] (August 2015).

[40] neural-style "Torch implementation of neural style algorithm".

[41] Alec Radford, Luke Metz, and Soumith Chintala(2015): Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. arXiv:1511.06434 [cs] (November 2015).

[42] Vijay Badrinarayanan, Kendall, and Roberto Cipolla(2015): SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation. arXiv preprint arXiv:1511.00561 (2015).

[43] SegNet Demo page.

[44] Volodymyr Mnih et al(2015): Human-level control through deep reinforcement learning. Nature 518, 7540 (2015), 529 – 533.

[45] David Silver et al(2016): Mastering the game of Go with deep neural networks and tree search. Nature 529, 7587 (2016), 484 – 489.

O'REILLY®
オライリー・ジャパン

深度学习入门：基于Python的理论与实现

从零开始，简明易懂，入门书的不二之选！

本书有什么特色？

- 使用Python 3，尽可能少地依赖外部库或工具，从零创建一个深度学习模型。
- 提供可运行的Python源代码以及可以让读者亲自试验的学习环境。
- 结合直观的插图介绍神经网络和深度学习的基础理论，简明易懂。
- 深入浅出地介绍误差反向传播法、卷积神经网络等看似复杂的技术。
- 介绍学习率的确定方法、权重的初始值等深度学习相关的实用技巧。
- 介绍Batch Normalization、Dropout、Adam等新方法并进行实现。
- 介绍自动驾驶、图像生成、强化学习等方面的应用。
- 介绍为什么加深层能提高识别精度、为什么隐藏层很重要等“为什么”方面的问题。

“对于非AI方向的技术人员，本书将大大降低入门深度学习的门槛；对于在校大学生、研究生，本书不失为学习深度学习的一本好教材；即便是在工作中已经熟练使用框架开发各类深度学习模型的读者，也可以从本书中获得新的体会。”

——摘自本书译者序

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议 计算机／人工智能

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Japan, Inc.授权人民邮电出版社出版

ISBN 978-7-115-48558-8
9 787115 485588 >
ISBN 978-7-115-48558-8
定价：59.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱:
ebook@turingbook.com。

在这可以找到我们:

微博 @图灵教育 : 好书、活动每日播报
微博 @图灵社区 : 电子书和好文章的消息
微博 @图灵新知 : 图灵教育的科普小组
微信 图灵访谈 : ituring_interview，讲述码农精彩人生
微信 图灵教育 : turingbooks